AFIT/GCS/ENG/91D-03

PARALLEL IMPLEMENTATION OF VHDL SIMULATIONS ON THE INTEL iPSC/2 HYPERCUBE

THESIS

Ronald C. Comeau
Captain, USAF

AFIT/GCS/ENG/91D-03

AFIT/GCS/ENG/91D-03

PARALLEL IMPLEMENTATION OF VHDL SIMULATIONS ON THE INTEL iPSC/2 HYPERCUBE

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science (Computer Science)

Ronald C. Comeau

Captain, USAF

December, 1991

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
PARALLEL IMPLEMENTATION OF VHDL SIMULATIONS ON THE INTEL iPSC/2 HYPERCUBE

5. FUNDING NUMBERS

6. AUTHOR(S)
Ronald C. Comeau

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/91D-03

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DARPA (LTC John Toole)
3701 N. Fairfax Dr.
Arlington, VA 22203

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

VHDL models are executed sequentially in current commercial simulators. As chip designs grow larger and more complex, simulations must run faster. One approach to increasing simulation speed is through parallel processors. This research transforms the behavioral and structural models created by Intermetrics' sequential VHDL simulator into models for parallel execution. The models are simulated on an Intel iPSC/2 hypercube with synchronization of the nodes being achieved by utilizing the Chandy-Misra paradigm for discrete-event simulations. Three eight-bit adders, the ripple-carry, the carry-save, and the carry-lookahead, are each run through the parallel simulator. Simulation time is cut in at least half for all three test cases over the sequential Intermetrics model. Results with regard to speedup are given to show effects of different mappings, varying workloads per node, and overhead due to output messages.

14. SUBJECT TERMS
Simulation, Parallel Processing, Discrete Event Simulation, VHDL (VHSIC Hardware Description Language)

15. NUMBER OF PAGES
161

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements.***

Block 1. Agency Use Only *(Leave blank)*

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s) Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes Enter information not included elsewhere such as. Prepared in cooperation with..., Trans. of..., To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only).*

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

*U S GPO 1990 0-273-271

Acknowledgements

I have so many people to thank, but if I said all that I wanted to about each person, I would have to make it another chapter. Therefore, excuse the brevity, but know that the feelings are sincere.

First, I wish to thank my family. My parents showed me that there is joy in learning as well as profit. They are responsible for my "thirst for knowledge" without which I could never have "reached for the gold that's afar." My family has been very supportive and encouraging throughout this eighteen-month eternity. To all my sisters, in-laws, and relatives, I say thank you.

Next, I wish to thank my friends at AFIT. Although the members of the GCS and GCE classes are competitive, they know that no one understands everything. The spirit of helpfulness, compassion, and cooperation between these students created a synergistic environment. That environment enabled us all to excel a little further than we would have were we on our own. They are simply the finest officers I have had the pleasure to meet.

I could not have completed this thesis without the patience and assistance of the faculty at AFIT. Maj Kanzaki sincerely believed that parallel VHDL was possible, even when I came to three dead-ends during this effort and "knew" that it would never happen. His faith taught me not only that I can be wrong (this was the first time I had ever been wrong), but that something can be learned from a failed attempt as well. I appreciate Dr Hartrum patiently explaining Chandy-Misra again and again and again. His insights into parallel processing helped me to realize several of the components of the parallel VHDL simulator. Maj Bailor, Maj Hobart, Maj Christianson, and Rick Norris never flinched when

I asked simplistic questions of them, they just held back a smile and answered the question -- thanks!

Most of all, I owe a world of thanks to my intelligent, beautiful, understanding, encouraging, ego-building, caring, hard-working wife, Kelly, and my wonderful, joyful, entertaining, smart son, Josh. Whenever I came home late and headed immediately to the PC, one of them would stop me to share the happenings of the day. It served to remind me that there is a world outside of AFIT and that a good grade-point average is a poor substitute for a happy family. Thanks for keeping me in touch with the most important two people in my life -- you two!

Table of Contents

Page

Preface ii

Table of Contents iv

List of Figures ix

List of Tables xiii

Abstract xv

I. Introduction 1-1

1.1 Overview 1-1

1.2 Background 1-1

1.3 Sequential Simulation Problem 1-2

1.4 Assumptions/Limitations 1-3

1.5 Research Objectives 1-4

1.6 Approach 1-4

1.7 Summary 1-6

1.8 Organization of Thesis 1-6

II. Background 2-1

2.1 Parallel Simulations 2-1

2.1.1 Conservative Approach 2-1

2.1.2 Asynchronous Distributed Approach 2-5

2.1.3 Time Warp 2-6

Page

2.1.4 Lookahead 2-7

2.1.5 Demand-driven Simulation 2-8

2.1.6 Summary of Approaches 2-10

2.2 VHDL 2-11

2.2.1 Definitions 2-11

2.2.2 The VHDL System 2-12

2.2.3 Maintenance Cycle with VHDL 2-14

2.3.4 Other VHDL Parallelization Efforts 2-16

2.3.5 Summary of VHDL 2-16

2.3 Summary 2-17

III. Parallel Simulator Design 3-1

3.1 Introduction 3-1

3.2 An Example 3-1

3.3 Analysis of Sequential VHDL 3-4

3.3.1 Simulator Organization 3-4

3.3.2 The Intermetrics VHDL Simulator 3-7

3.4 Approaches to Parallel VHDL Simulation 3-8

3.5 Data Structures 3-9

3.6 Process of Simulation 3-15

3.6.1 Simulation Cycle 3-15

Page

3.6.2 The VHDL Test Bench 3-19

3.7 The iPSC/2 Hypercube 3-20

3.8 Parallel Simulation Issues 3-21

3.8.1 The Parallel Design 3-21

3.8.2 Chandy-Misra Paradigm 3-23

3.8.3 Internode Dependence 3-25

3.8.4 Eliminating Unnecessary Features 3-27

3.9 The Simulation 3-28

3.10 Summary 3-31

IV. Implementation 4-1

4.1 Introduction 4-1

4.2 Editing Intermetrics-Generated C Source 4-1

4.2.1 Capturing C Source Code 4-1

4.2.2 Editing the Big C Source File 4-5

4.2.3 Compiling on the Hypercube 4-12

4.3 Simulation Subroutines 4-13

4.3.1 Subroutines From Sequential Models 4-14

4.3.2 Useless But Necessary Subroutines 4-14

4.3.3 Original Subroutines 4-15

4.4 Concerns for Parallel Simulation of VHDL 4-20

Page

4.5 Mapping to the iPSC/2 4-24

4.6 Summary 4-26

V. Test Cases 5-1

5.1 Introduction 5-1

5.2 The Carry-Save Adder 5-1

5.3 The Ripple-Carry Adder 5-2

5.4 The Carry-Lookahead Adder 5-2

5.5 Logical Processes 5-9

5.6 Output From the Parallel Simulator 5-10

5.7 Verification 5-15

5.8 Summary 5-16

VI. Experimentation and Analysis 6-1

6.1 Introduction 6-1

6.2 Experimental Options and Constraints 6-1

6.3 Comparisons 6-2

6.4 Analysis of Results 6-12

6.5 Recommendations for Future Research 6-13

6.6 Conclusions 6-15

Page

Appendix A. PVSIM.C Source Code A-1

A.1 PCSIM.C A-1

Appendix B. An Example B-1

B.1 The Full Adder B-1

Appendix C. Statistics C-1

C.1 The Ripple-Carry Adder C-1

C.2 The Carry-Lookahead Adder C-2

C.3 The Carry-Save Adder C-2

Appendix D. Configuration D-1

D.1 The Files D-1

Appendix E. Release E-1

E.1 Authorization E-1

Vita VITA-1

Bibliography BIB-1

List of Figures

Figure — Page

2.1. An Occurrence of Deadlock 2-2

2.2. Another Occurrence of Deadlock 2-3

2.3. Simple Circuit with Feedback 2-9

2.4. Phase Cost Comparison 2-15

3.1. VHDL Code for Full Adder Entity 3-2

3.2. Full Adder Circuit Diagram 3-2

3.3. VHDL Entity and Behavioral Descriptions for an AND Gate 3-3

3.4. Structural Description of a Full Adder 3-3

3.5. Configuration of a Full Adder 3-4

3.6. Simulator Organization 3-5

3.7. Full Adder Circuit Diagram 3-6

3.8. Intermetrics' Organization 3-8

3.9. Full Adder Circuit Diagram 3-9

3.10. Signal Record Data Structure 3-10

3.11. Behavior Instance (BI) Data Structure 3-11

3.12. Code Excerpt From AND Routine 3-11

3.13. Active Record Data Structure 3-12

3.14. Interrelationship of VHDL Simulation Data Structures 3-14

3.15. The VHDL Simulation Cycle 3-15

3.16. Pseudocode for Sequential VHDL Simulation 3-19

Page

3.17. VHDL Source Code for the Full Adder Test Bench 3-20

3.18. Simulation Model for Two Nodes 3-22

3.19. Full Adder Circuit Diagram 3-26

3.20. Configuration of a) Half Adder and b) Full Adder 3-28

3.21. Ripple-Carry Adder Configuration 3-29

3.22. Carry-Lookahead Adder Configuration for a) 4 bits, and b) 8 bits 3-30

3.23. Carry-Save Adder Configuration 3-31

4.1. Sample Compilation Script File 4-2

4.2. Example User Session 4-3

4.3. Build Shell File 4-4

4.4. Code Excerpt for Steps 1 and 2 4-7

4.5. Edited Code Excerpt for Steps 1 and 2 4-7

4.6. Code Excerpt for Steps 2 and 3 4-8

4.7. Edited Code Excerpt for Steps 2 and 3 4-8

4.8. Code Excerpt for Steps 1 4, 5, and 6 4-8

4.9. Edited Code Excerpt for Steps 4, 5, and 6 4-9

4.10. Code Excerpt for Steps 6 and 7 4-9

4.11. Edited Code Excerpt for Steps 6 and 7 4-9

4.12. Code Excerpt for Steps 8 and 9 4-10

4.13. Edited Code Excerpt for Steps 8 and 9 4-10

4.14. Code Excerpt for Step 10A 4-10

Page

4.15. Edited Code Excerpt for Step 10A 4-10

4.16. Code Excerpt for Step 10B 4-11

4.17. Edited Code Excerpt for Step 10B 4-11

4.18. Full Adder Makefile 4-12

4.19. Parallel Simulation Pseudocode 4-19

4.20. Example Circuit for Parallel Simulation 4-22

4.21. MAPPING Output Example 4-25

5.1. Approach 1 for Carry-Lookahead Partitioning 5-3

5.2. Approach 2 for Carry-Lookahead Partitioning 5-5

5.3. Approach 3 for Carry-Lookahead Partitioning 5-6

5.4. Approach 4 for Carry-Lookahead Partitioning 5-7

5.5. Sample of Intermetrics' Report 5-11

5.6. Sample Output from an Eight Node Run 5-12

5.7. Sample Output from an Eight Node Run After Sorting 5-14

6.1. Comparison of Applications 6-3

6.2. Speedup of Applications 6-4

6.3. Comparison of Effect of Test Vectors 6-6

6.4. Test Vector Effect on Speedup 6-7

6.5. Effect of Spin Loops on CLA 6-8

6.6. Speedup in the CLA From the Spin Loop Effect 6-9

6.7. Overhead in Processing Output Statements 6-10

Page

6.8. Speedup for Output Options 6-11

6.9. Active Signal Comparison 6-12

B.1. Full Adder Circuit Diagram B-1

B.2. Current Processor States B-3

B.3. Current Active Records B-3

B.4. Current Processor States B-3

B.5. Current Active Records B-4

B.6. Current Processor States B-4

B.7. Current Active Records B-6

B.8. Current Processor States B-6

B.9. Current Active Records B-7

B.10. Current Processor States B-7

List of Tables

Table Page

3.1. Examples of Signal Records 3-10

3.2. Examples of Behavioral Instance Records 3-12

3.3. Active Record Example 3-13

3.4. Behavior List 3-16

3.5. Current Active Records 3-18

4.1. Intermetrics' Generated Files 4-5

4.2. Parallel Simulation Routines 4-13

5.1. Typical Execution Times for Approach 1 5-3

5.2. Typical Execution Times for Approach 2 5-4

5.3. Typical Execution Times for Approach 3 5-5

5.4. Typical Execution Times for Approach 4 5-7

5.5. Statistics for the Four Approaches 5-8

5.6. Comparison of the Four Approaches 5-9

6.1. Statistics on Test Cases 6-2

6.2. Statistics on Test Vector Test Cases 6-6

B.1. Lp.arcs File B-1

B.2. Queue.dat File B-1

B.3. Signal Records at Initialization B-2

B.4. Behavioral Instance Records B-2

Page

B.5. Node 0's Signal Records at Time 50 ns B-5

B.6. Node 1's Signal Records at Time 53 ns B-7

Abstract

VHDL models are executed sequentially in current commercial simulators. As chip designs grow larger and more complex, simulations must run faster. One approach to increasing simulation speed is through parallel processors. This research transforms the behavioral and structural models created by Intermetrics' sequential VHDL simulator into models for parallel execution. The models are simulated on an Intel iPSC/2 hypercube with synchronization of the nodes being achieved by utilizing the Chandy-Misra paradigm for discrete-event simulations. Three eight-bit adders, the ripple-carry, the carry-save, and the carry-lookahead, are each run through the parallel simulator. Simulation time is cut in at least half for all three test cases over the sequential Intermetrics model. Results with regard to speedup are given to show effects of different mappings, varying workloads per node, and overhead due to output messages.

I. Introduction

1.1 Overview

Very Large Scale Integration (VLSI) electronic chip designs have grown to the point where designers can neither afford the time nor the money to bread board very complex designs. Since designers cannot test hardware prototypes, they must be satisfied with testing a simulation of their designs. However, because they are working with a simulation, tests can be quickly reconfigured, another simulation run, and the circuit redesigned before manufacturing any expensive hardware (Vyas:402). Thus, the simplicity and effectiveness of documenting and testing chip designs, by taking advantage of the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL), result in a better design for the chip.

Nevertheless, chip designs are now so complex that sequential VHDL simulators offer only limited capability as a useful tool. One approach to handling the complexity of a design is to distribute the simulation of the design over several processors. By mapping VHDL's capabilities to a parallel processor, the chip designer can still receive results from simulations in a reasonable amount of time, even for complex circuits. Currently, such a capability does not exist.

1.2 Background

The Department of Defense (DOD) has identified VHDL as the standard language for all designs of Application Specific Integrated Circuits (ASICs) (MILS:64-4). Using VHDL, the DOD specifies the desired functional behavior of a desired chip, with respect

to time (i.e., a "black box"). Competing contractors are then able to provide VHDL structural designs that accomplish the desired behavior. It is the structural design which specifies the actual hardware components that perform the function described by the behavioral model. Additionally, since well-written VHDL code is readable by both machines and humans, it is self-documenting (VLRM:i) (Lips:xi).

As a simple example, suppose the DOD specifies a requirement for a chip that takes a 32-bit integer as its input, and after a delay of five nanoseconds (ns), produces the number's square-root. Contractors who possess the necessary skill and technology describe the chip's subcomponents in VHDL and run simulations. Once they verify that the VHDL design meets the specified requirements, they submit the VHDL code as their structural design. DOD tests the VHDL designs from each contractor to confirm they meet the minimum requirements and to verify the results. Based on the outcome of those tests, DOD chooses a chip design and awards the contract.

Since government contractors use VHDL as a common platform, DOD can directly compare different designs to each other and select the best design. Furthermore, since VHDL is standardized, designs for interacting components are thus compatible with each other, despite the company that originally created them. This compatibility allows designs to be scaled up or scaled down. The resultant language is of great benefit in the phases of design research, testing, and design maintenance (Lips:xii).

1.3 Sequential Simulation Problem

As the size and complexity of VLSI circuit designs have grown, simulations of the

designs have become increasingly complex, requiring faster processing. One approach to gaining the necessary computing power is to use parallel processors. Mapping VHDL simulations to a parallel processor offers speedup for even more complex circuit designs than those that are currently simulated sequentially. Therefore, the goal of this thesis is to develop and test a VHDL compiler design that enables sequential VHDL models to be mapped to and run on a parallel processor.

1.4 Assumptions/Limitations

Before tackling the sequential simulation problem, the environment in which parallel simulations are to evolve must be defined. Therefore, below are assumptions about that environment and the constraints that it imposes:

1) This thesis uses the terms "parallel" and "distributed" interchangeably throughout. Usually a "distributed processor" implies a multicomputer that is made up of processors that are not geographically close to each other (Akl:18). A "parallel computer" is a computer with multiple processors that work simultaneously on subproblems of a large problem. One can solve the original, large problem by combining the results from each subproblem (Akl:2). Since the definition of "close" varies from author to author, "parallel" and "distributed" are used interchangeably.

2) The distributed processor used for development and research is the Intel iPSC/2 Hypercube with Release 3.2 as the system software.

3) Source code is written in the standard C programming language (non-ANSI) and is compiled using the Green Hills C compiler.

4) To further research efforts for both DARPA and AFIT and stay consistent with the AFIT environment, a variation of the Chandy-Misra scheduling algorithm for event-driven simulations is used.

5) The output from the analyze, model generate, and build phases of the Intermetrics VHDL compiler are correct and accessible.

6) The VHDL test cases are within the VHDL subset that is used to demonstrate parallelized VHDL.

1.5 Research Objectives

The main objective is to parallelize standard VHDL behavioral and structural simulations. So that others may use the same process to design even more complex VHDL parallel simulations requires clear documentation. Additionally, the three test cases provide a common set of VHDL simulations for future students to use as a baseline in their parallel processing research.

1.6 Approach

The first step is to understand how the sequential VHDL simulator works. It analyzes VHDL code, processes it into an intermediate format code, then transforms the intermediate code into C source code. A sequential computer then compiles and executes the C source code.

To simulate VHDL in parallel, commands for the parallel processor must be inserted into the code at some point in the process. C source code is much more readable and easier

to work with than the intermediate format. Therefore, our approach is to intercept, analyze, and transform the C source code for parallel execution.

There are two ways to transform the code. The first is alter the C source code as it is generated, by changing the Ada and C files that make up the sequential VHDL system. The second approach is to run a set of VHDL code through the system and analyze the generated C source code files.

To ensure progress, the problem has been attacked from both directions. Using only the first method, a particular statement might have been overlooked while sifting through the 14 Mbytes of lines of code. However, depending only on the second method might have limited the subset of VHDL too much, thus, rendering the compiler less useful for later students. Combining the two approaches results in a much more universal subset of VHDL with which to work. It also provides a better understanding of the overall VHDL process.

After choosing an adequate subset of VHDL, a compiler that translates sequential C source code to parallel C source code is needed. Testing the transformed code requires the development and implementation of a parallel VHDL simulator. The simulator takes the transformed VHDL descriptions as input, runs the VHDL simulation in parallel, and produces output statements that correspond to signal changes.

After designing the compiler, three test cases have been run through it to insure the functionality of the compiler. These test cases are the structural architectures for a ripple-carry adder, a carry-lookahead adder, and a carry-save adder. The successful runs of these tests prove that parallelizing at least a subset of VHDL is possible. Final analysis of the tests includes lessons learned, suggestions for improvements, and recommendations for

further studies.

1.7 Summary

Because VLSI chip designs are becoming more complex, it is unrealistic for designers to build and test a prototype of a microelectronic circuit. Therefore, today's chip designers use VHDL to submit chip designs to the DOD. These designers are discovering that sequential execution limits the speed at which a VHDL simulation can be run. By distributing VHDL throughout a parallel processor, designers complete a simulation run more quickly.

If each individual simulation is faster, designers are able to perform more tests on different configurations of the circuit. These easily reconfigured circuits and tests result in a more robust chip design. Increasing the reliability of each chip increases the reliability and accuracy of the weapon systems which use these chips. More reliable weapon systems increase the ability to deliver the payload to the intended target and avoid collateral damage.

1.8 Organization of Thesis

Chapter two contains background material on parallel simulation and VHDL. The parallel simulation discussion includes a comparison of different types of discrete-event paradigms such as Chandy-Misra and Time Warp. The VHDL section gives a brief review of VHDL and points out the benefits of VHDL simulations.

In the third chapter, the reader is introduced to the process of designing the parallel VHDL simulator. First, the Intermetrics sequential simulator is analyzed and modeled.

Then, design of the parallel simulator begins. Finally, concerns inherent in parallel VHDL simulation are addressed.

Chapter four covers implementation of the parallel simulator. Specifically, it provides direction to repeat this work with other VHDL models. Several examples are shown to help the reader understand parallel simulation concepts.

In Chapter five, the reader can find three test cases. These test cases give the user a hands-on insight into the problems of parallel simulation. Also in chapter five is a discussion about output verification.

Lastly, Chapter six presents findings from running three test cases through the parallel simulator. Results from various experiments are compared and explanations submitted for possible causes of any interesting phenomena. It also offers some conclusions for this research and recommends potential areas for future work.

II. Background

To develop parallel simulations for VHDL, requires familiarity in two distinct areas. The first is simulations in general. One must understand simulation approaches to choose which would be best for modeling the behavior of an electronic circuit. The second area is VHDL itself. One cannot parallelize VHDL without comprehending the interdependencies of sequential VHDL's separate stages.

2.1 Parallel Simulations

2.1.1 Conservative Approach To help parallelize VHDL simulations, Chandy and Misra suggest looking at the simulations as a series of discrete events. Every discrete event must have a simulation time-stamp attached. The user assigns electronic components, called logical processes (LPs), to one of several processors. Each processor must be aware of which LPs are assigned to which processors and the dependencies between LPs. After an LP finishes processing new inputs, it sends a message to all downstream LPs. The message contains its updated output value and the simulation time at which the update occurred (Chan:199). Since all of the LP's can be working simultaneously on several processors, the VHDL simulation has been parallelized over the sequential implementation and a corresponding speedup is expected.

One drawback to this approach is that deadlocks can occur. Figures 2.1 and 2.2 each depict a possible scenario for deadlock in a simulation. One case of deadlock occurs when a processor is waiting for an input from an upstream processor that never arrives, as shown in Figure 2.1. "NE Time" is the simulation time for the next event in that LP. The time

shown next to the arrow between LPs is the simulation time of the last message sent along that path.

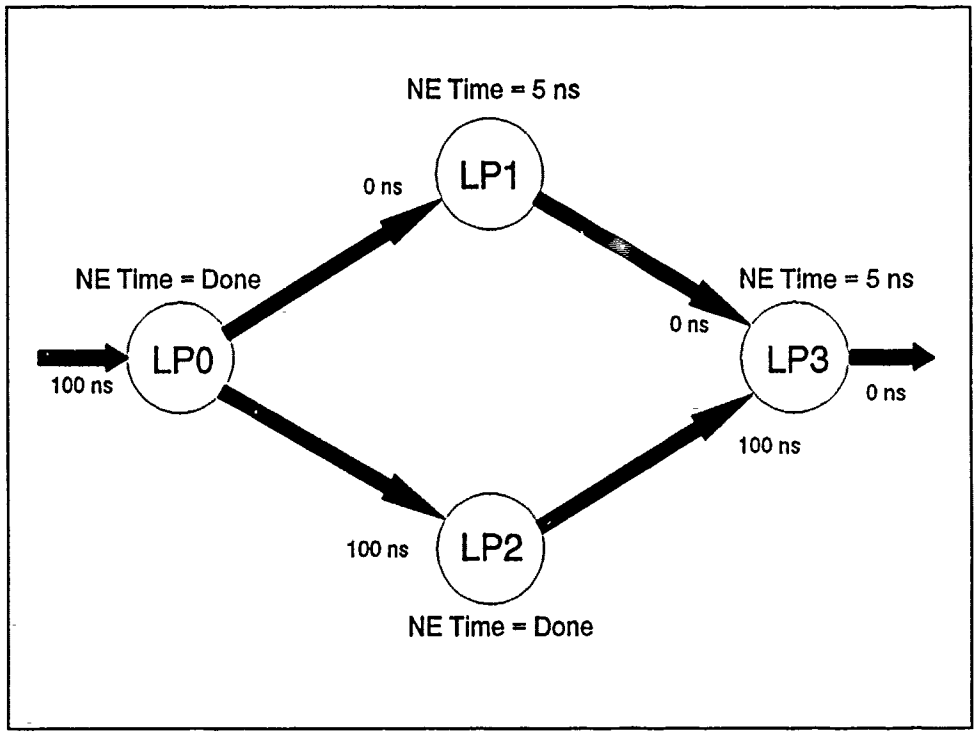

Figure 2.1 An Occurrence of Deadlock.

Suppose LP0 never sends a message to LP1. Then LP1 waits at simulation time 0 ns even after LP0 and LP2 are completely done with the simulation. Since LP3 must wait for a message from LP1 before proceeding, LP1 effectively blocks LP3 and any downstream processes that are waiting for a message from LP3. Thus, since there is nothing in the system that will clear the blocked LPs, there is a deadlock.

Another example of deadlock can occur when there is feedback among the LPs, as shown in Figure 2.2. Chandy and Misra call this type of deadlock "cyclic waiting" since the cycle must be broken before the processes can continue (Misr:55). In Figure 2.2, LP0 has

received a message from an upstream LP (not shown) at simulation time 10 ns. Also LP0 has never received a message from LP2. Therefore, the simulation time along that arc is still at 0 ns.

LP0 cannot proceed until it receives a time from LP2 of at least 10 ns. LP2 is waiting for LP1 to send a message for time 100 ns or greater. But LP1 is waiting for LP0. Since each LP is waiting for another, deadlock occurs again.

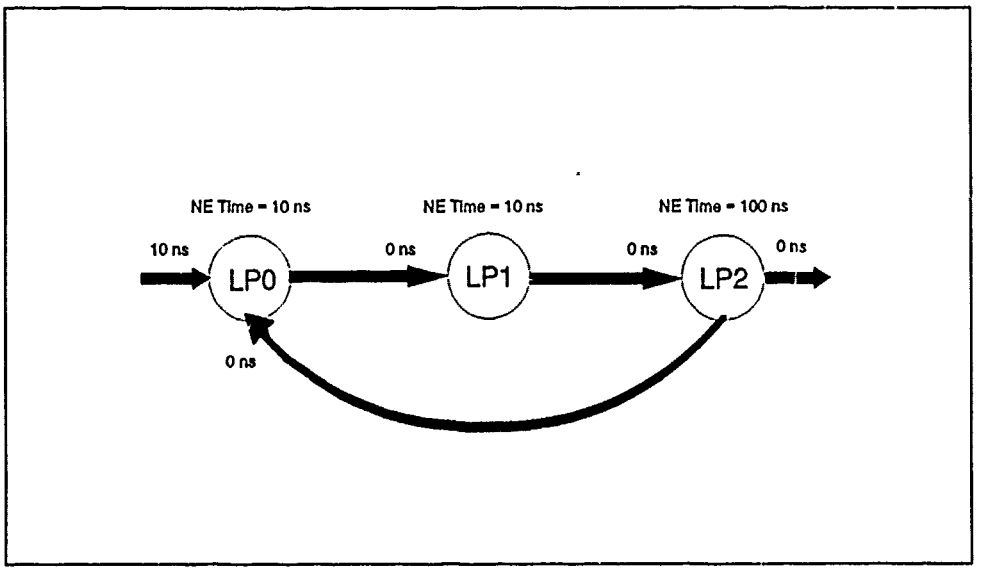

Figure 2.2 Another Occurrence of Deadlock.

Although Chandy and Misra admit deadlock is a problem, they submit that deadlock will happen very infrequently and, therefore, is only a minor concern (Chan:203-204). One proposed solution for deadlock is for the controlling processor to poll all of the other processors until it finds out what the next simulation time should be. Each processor responds with its earliest next event time. The controller analyzes all of the responses from processors. It chooses the lowest time, called a "safe" time, and broadcasts this simulation

time to each processor. Each processor updates their simulation clock to this time allowing at least one of them to restart. Restarting one processor breaks the deadlock, and the simulation can continue (Chan:204).

For example, in Figure 2.1, the controller finds the lowest next event time to be 5 ns. It broadcasts this time to all LPs. That broadcast effectively updates every input arc time to 5 ns if the current value on that arc is less than 5 ns.

In Figure 2.2, the controller breaks the deadlock by broadcasting 10 ns, which enables LP0 and LP1 to continue. Obviously, the controller must intervene constantly to recover from frequent deadlocks. Therefore, Chandy and Misra suggest an alternative.

That alternative is to use "NULL messages" to update the simulation time of every dependent LP. Although these messages provide no data information for the simulation, they do ensure progress for all downstream LPs. A NULL message is sent in two cases: 1) when an LP sends out a message on one output arc, it must send out a NULL message on every other output arc for that same time; and 2) when an LP receives an updated time from an upstream LP, it adds its propagation delay to this new time and sends the modified time out to all of its downstream LPs.

For example, consider the problem in Figure 2.1 once more. If LP0 sends a message to LP1 every time it sends a message to LP2, then LP1 is able to proceed. Since LP1 is processing, LP3 is able to proceed as well.

In Figure 2.2, assume the delay through each LP (which is known beforehand) is 5 ns. LP0 sends a NULL message for time 5 ns (0 ns + 5 ns) to LP1. After receiving the NULL message, LP1 sends a NULL message for time 10 ns (5 ns + 5 ns delay) to LP2.

LP2 then sends a NULL message for time 15 ns (10 ns + 5 ns delay) to both of its downstream processes, including LP0. LP0 now has 10 ns and 15 ns on its input arcs, allowing it to process its next event. Thus, this alternative avoids deadlock.

Chandy and Misra believe such deadlock predicaments are rare occurrences. They submit that the overhead of a central controller or of sending and receiving NULL messages is worth the assurance of deadlock recovery or avoidance. These approaches prevent a logical process from proceeding until it is guaranteed not to receive a message for a past simulation time. Thus, it is called the "conservative" approach.

2.1.2 Asynchronous Distributed Approach The asynchronous distributed approach differs from Chandy and Misra's proposal in that it prevents the possibility of deadlock. By preventing deadlock, recovery from deadlock is never an issue. This approach allows more processing time to be focused on the simulation itself.

Ghosh and Yu propose that deadlock can be prevented. Every component (LP) sends out a message with the time-stamp of the lowest of its input times plus the propagation delay through the component. Every LP that is waiting for a signal from the component receives this time-stamped message (Ghos:76). If the value of the output does not change, the message may be considered a "null message."

If every LP broadcasts null messages, there is no possibility of deadlock (Ghos:77). However, Ghosh and Yu admit that when simulating circuits with feedback loops, LPs might spend a large amount of time processing null messages. Unfortunately, most of these null messages will not advance the simulation at all. Wasting time processing useless null

messages is the price that one must pay to insure that the system remains deadlock-free.

2.1.3 Time Warp Jefferson introduced another simulation paradigm, called Time Warp. Jefferson criticized the Chandy-Misra paradigm because the downstream processors in a pipeline parallel execution may sit idle while the upstream processors are busily computing. A "blocked" logical process is one that sits idle, waiting until it can safely continue processing. "Blocked" is also used to describe the processor's state when the processor only contains idle logical processes.

Instead of blocking a process, Jefferson proposes to let each logical process proceed at its own pace based on present information. If a message comes in with a time stamp in the past, then the logical process must roll back to that simulation time to process the message (Jeff:411).

A further complication of rolling back a process is the fact that it may have sent out messages to other processes that now contain incorrect data. The downstream processes may have used the incorrect data to send out their own messages, and so on. Jefferson acknowledges that any message sent out in error must now be cancelled. An "antimessage" is the mechanism he uses to cancel an errant message (Jeff:414).

Since a logical process may have to rollback to some previous point in the simulation, every state the process has been in must be saved. For hundreds or thousands of logical processes, such memory overhead is unacceptable. Therefore, Jefferson introduces a "global virtual time" (GVT) which guarantees the state of all processes are correct at that time. After saving the state of every logical process for the GVT, all previous state information

can be discarded, thus freeing valuable memory (Jeff:417-419). Thus, one trades the overhead of blocking or deadlock recovery/prevention for the overhead of saving states, calculating GVTs, and rolling back.

2.1.4 Lookahead Fujimoto proposes using "lookahead" to calculate likely, future simulation outputs without actually implementing the event. This approach combines both Jefferson's Time Warp algorithm and the Chandy-Misra paradigm because a processor spends its idle time calculating probable future events (outputs). However, the processor does not apply those outputs until the event time matches the simulation clock. Since the processor does not send any messages until the simulation time is safe, processors never have to be rolled back.

Although the lookahead approach is attractive, Fujimoto believes that one LP's dependence on another inherently limits some parallel applications as to how they can be parallelized. He formally defines the limiting factor as "lookahead" and claims it plays a critical role in simulator performance (Fuji:36). Lookahead is the ability of a logical process to schedule events that will occur in the future.

For example, assume a two-input component (with IN0 and IN1) produces a resultant output 10 ns after receiving an input. Also assume that component receives a new input on IN0 with time-stamp t_1, then the result from the new input will be produced at time $t_1 + 10$ ns. Thus, the component (and its corresponding LP) has a lookahead of 10 ns.

A problem occurs, however, if this LP receives another message on its other input line IN1 with a time-stamp of t_0, where $t_0 < t_1$. First, it throws away the previous result

calculated for t_1 + 10 ns. Then it recalculates its output at t_0 + 10 ns and its output at t_1 + 10 ns. Finally, the LP sends out a message for each of these new events. In this scenario, the Chandy-Misra paradigm is more effective since useless calculations would be avoided.

However, suppose input line IN0 changes often while input line IN1 only changes occasionally. The LP is now sits idle for most of the time waiting for IN1 to give it a "safe" time (as defined in section 2.1.1). Fujimoto claims this type of situation is typical of a VHDL simulation. He further claims that such simulations are inherently limited in how much speedup they can gain from parallel execution (Fuji:39-40).

2.1.5 Demand-driven Simulation Subramanian and Zargham agree with Fujimoto that the amount of parallelism one can do with discrete-event simulation is limited. However, they point out that instead of reacting to input changes, the problem can be looked at in reverse and the value of the output at time T can be observed. The processor traces the output signal at time T backward through the circuit, subtracting off the processing delay of each component through which the signal passes.

Ultimately, the signal will be traced back to the system input signals at a particular time. The processor already knows the system input values at any given time. Therefore, the input values are traced forward through the circuit until the the processor can calculate the final output for the desired time. Usually, the demand-driven approach does not use every LP to calculate the requested output. Thus, the processor avoids calculating redundant and useless information. Thus only the information desired is processed, when it is needed (Subr:486-487).

There are two drawbacks to the demand-driven approach. First, one does not run a simulation to get expected results. One runs a simulation to see if there are unexpected results. Using only the demand-driven simulation paradigm, unexpected effects are masked at the requested output times; thus, they slip by undetected.

Second, if there is feedback in the circuit, quite a bit of useless calculation could occur. Suppose the simple circuit shown in Figure 2.3 is simulated and that the delay through the circuit is 5 ns. Suppose, further, that a '0' is applied to the signal labeled "IN" at 0 ns and a '1' is applied at 95 ns. Accordingly, an output of '1' is expected at 100 ns on the signal with the "OUT" label.

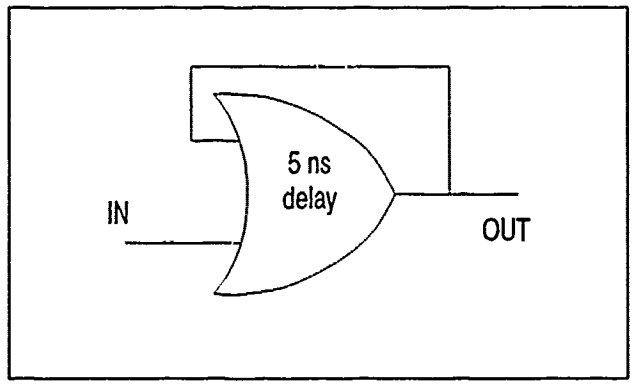

Figure 2.3 Simple Circuit with Feedback

Following the paradigm, the computer requests the output value at 100 ns. Subtracting the circuit delay, the computer inquires for the values of the inputs at 95 ns (100 ns - 5 ns delay). Since one input is the OR gate's output, the computer inquires for the output value of the circuit at 95 ns, which causes an inquiry for the input values at 90 ns, and so on. This cycle continues until the computer traces the signals back to the beginning of

the simulation. Thus, instead of performing one calculation as in conventional simulation algorithms, 20 calculations are performed on this simple circuit for a very short simulation run. Extrapolating this example to a simulation composed of millions of subcomponents, and simulated over several hours, clearly shows this paradigm to be unfeasible.

2.1.6 Summary of Approaches The question then is which approach runs discrete-event simulations more efficiently? It appears that choosing one approach over the others depends mostly on the application that one is trying to simulate (Proi:7-2). Because of limitations individual environments place on simulations, and differing requirements of specific applications, architectures, and mapping strategies, it is difficult to obtain a meaningful, unbiased comparison.

There exists a simulation scenario for all of the above proposals which shows each to give the best performance. However, there is no guarantee that any one simulation will be the only type run or even the most likely. Nic l identifies certain "rules of thumb" that have proven useful in executing discrete-event simulations (Nico:98):

1) Use Lookahead knowledge.
2) Avoid high message fan-out.
3) If high message fan-out cannot be avoided, then aggregate LPs onto fewer processors to bypass high communication costs.
4) Use uniform time-increment distributions.
5) Utilize hardware accelerators when using the Time Warp paradigm.

By following the general guidelines above, any simulation mechanism may be used effectively

for typical discrete-event simulations.

2.2 VHDL

2.2.1 Definitions The following definitions are taken directly from Eickmeier's research [Eick]. Eickmeier arrives at these definitions by combining the writings of several authors and the research environment at the Air Force Institute of Technology (AFIT). They work best for describing the VHDL environment used for this research as well.

Design Entity - The *design entity* is the primary hardware abstraction in VHDL. It represents a portion of a hardware design that has well-defined inputs and outputs and performs a well-defined function. A design entity may represent an entire system, a sub-system, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between. A design entity has two parts: an *entity declaration* and an *architectural body*.

Entity Declaration - Defines the entity's interface to the external environment; it specifies the *ports* of the entity in which data may flow in and out. Formally, ports may have a mode **in, out, inout,** and **buffer**, for showing the flow of data. The first three modes are self-explanatory. This research does not use the mode **buffer.** (VLRM) and/or (Lips) provide a complete description of this mode.

Architectural Body - The description of the internal behavior or structure of a design entity. A structural description decomposes the design entity into lower level entities, and describes the connections between these entities. A behavioral description is used at the lowest level of decomposition and shows how the entity transforms inputs to outputs. In this research, behavioral architectures describe logic gates in the system (e.g., AND, OR, XOR) and

structural architectures describe all higher-level components (e.g., half adder, full adder, ripple-carry adder).

Signal - An object that holds a value and directly corresponds to some type of metal interconnection within a circuit. The signal may change values throughout the simulation.

Port - A signal that appears in the interface list of an entity declaration. See the above definition for an entity declaration for more details.

Block - A design entity may be described in terms of a hierarchy of *blocks*, each of which represents a portion of the whole design.

External Block - The top-most block in a hierarchy. This block is the design entity itself, and it defines the interface of the design entity to the external environment.

Design Hierarchy - The result of successive decomposition of a design entity into components. It also binds those components to other design entities that may be decomposed in like manner. Taken together they represent a complete design. Such a collection of design entities is called a *design hierarchy*.

Model - A *model* is the elaboration of the design hierarchy in the VHDL simulation environment. The *model* is executed to simulate the behavioral or structural design represented by the *model*.

2.2.2 The VHDL System VHDL is called the "FORTRAN of hardware description languages" because of its wide acceptance as the standard language for hardware design (Nash:20). VHDL source code must go through several transformations before a simulation can be run. The first stage of the VHDL system is the compiler.

The compiler analyzes the syntax for correctness and performs a static semantic check of the VHDL source code. The lexical analyzer checks the VHDL syntax and passes tokens to the parser generator. It is the parser generator that performs semantic checking (Vyas:402). The semantic analyzer not only checks for errors but also gathers and stores information on any identifiers used in the source code.

Upon successful compilation, the compiler places the VHDL unit into the design library. The design library allows the user to compile VHDL entities separately and use them later without recompiling them (Coel:321). The compiler also translates the VHDL source into an intermediate form and later uses the intermediate form to create C code for executing the actual simulation. C was chosen because of its flexibility and its ability to manipulate machine-level data structures easily (Vyas:403).

Because the VHDL system is set up in the manner described above, it offers several benefits to the user. These benefits include separate compilation, strong typing, inclusion of time as part of the simulation, powerful semantic and syntactic features, and the ability to handle concurrent simulation of components (Coel:321, Vyas:401) (similar to how an operating system handles multitasking). Although the VHDL language allows concurrent simulation, the actual, sequential implementation on Von Neuman-type architectures does not. That is why this research is critical, since it will bring the concurrent execution abstraction to reality.

Separate compilation grants the user the ability to add entities to the library without recompiling lower-level components that make up the entity's behavior or structure (Vyas:401). Thus, separate compilation reduces overall recompilation time. Further,

because of VHDL's syntax, semantic rules, and strong typing, one can describe the behavior of a very complicated device in just a few lines of VHDL (Coel:322).

2.2.3 Maintenance Cycle with VHDL An electronic circuit has a three-phase life cycle associated with it: 1) design; 2) production; and 3) maintenance. Of these three, the costs associated with maintenance consist of about 60% of the circuit's life time costs (Wint:145) as shown in Figure 2.4. Put another way, for each chip, maintenance costs are roughly six times the cost required for design. Thus, using VHDL not only lowers costs during the design phase, but lowers overall maintenance costs as well.

The three main reasons cited by Winter and Lowenstein for nigh maintenance costs are a lack of computer processable data, parts obsolescence, and lack of cooperation between designers and maintainers (Wint:145). VHDL can help to overcome each of these barriers. First, chips designed with VHDL can be delivered with the VHDL source code. VHDL is easier to find and analyze than several volumes of hardcopy documentation and original design schematics. VHDL also can be processed by a computer, so that design data does not have to be reentered into a computer for the maintainer to run a simulation on the system.

Though the military may use a system for twenty years, the manufacturer typically stops production of the system's parts after three to seven years (Wint:146). However, by exploiting VHDL the military can easily contract with a manufacturer for an obsolete part to be produced. It is easy because the military will have the original specifications in VHDL, which can then be used by the manufacturer to produce the needed component. The

original specifications in VHDL assures the government of receiving the part and guarantees easy integration into the system. Therefore, DOD avoids time-consuming and costly research and reverse-engineering.

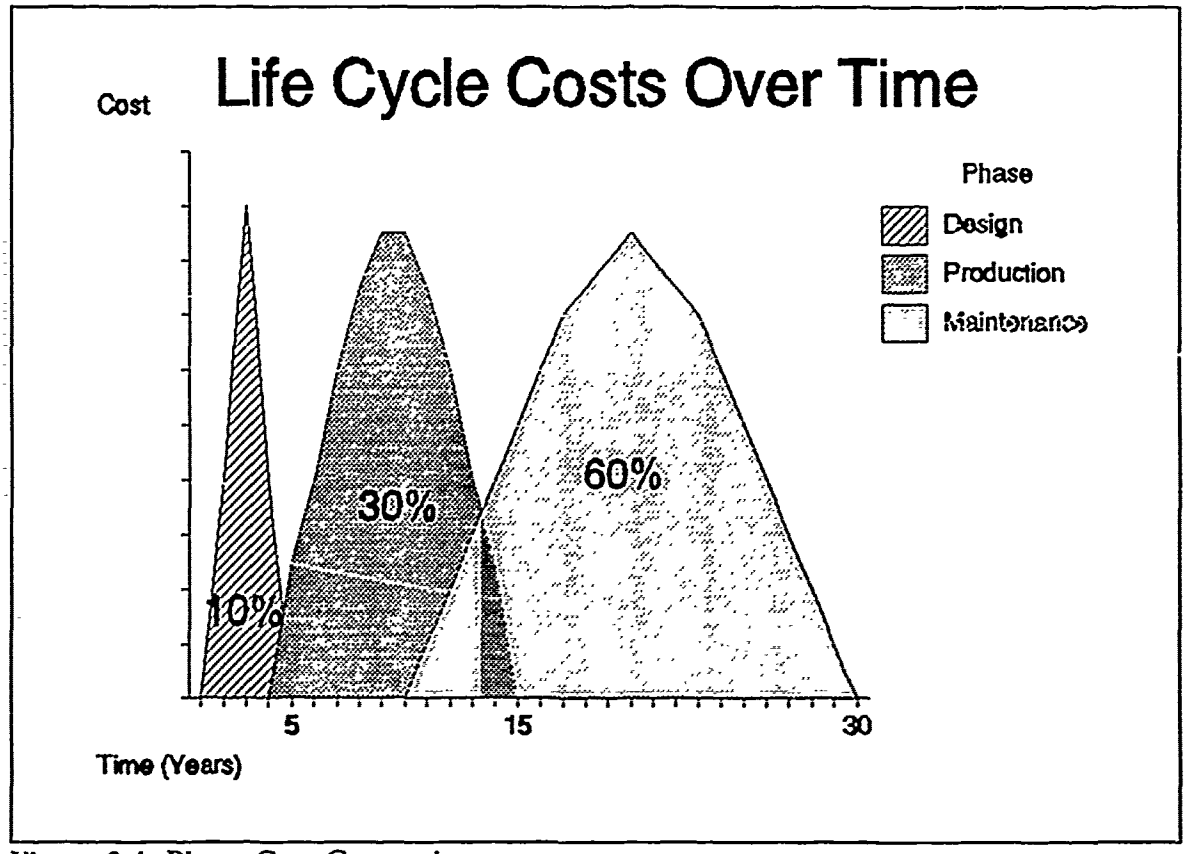

Figure 2.4 Phase Cost Comparison.

Finally, as the device passes from design to production to maintenance, any tests that have been developed can be passed along with the VHDL source code (Wint:150). This is important because currently each stage is developing their own set of tests, many of which are redundant when combined with the previous stage's test cases. Thus, a maintenance engineer immediately has a whole suite of tests to help diagnose any problems in the system without spending valuable time redeveloping these same tests.

VHDL designs also can transcend corporate borders to improve collaboration and interaction. Companies are able to use their individual specialties to deliver a high-quality, well-integrated component to the government (Waxm:310). IEEE's acceptance of VHDL as IEEE Standard 1076 is what makes cooperation within and between industry and government possible.

2.2.4 Other VHDL Parallelization Efforts Bhaskar attempted to parallelize VHDL at the microcode level in 1987 (Bhas:54). Their approach is to transform a sequential VHDL behavioral description into VHDL made up of statements that can all be executed in parallel. The algorithm analyzes a VHDL behavioral description and converts it into a process graph. Then, it optimizes the graph for parallel execution. Finally, it translates the graph back into VHDL source code (Bhas:54). They then translate the VHDL into MIMOLA to run the simulation. The key to parallelization lies in the process graph. Every statement within a node of the process graph have no dependencies between them. Therefore, they can be executed in parallel (Bhas:56). So far only preliminary results are available, but they are encouraging.

2.2.5 Summary of VHDL Using VHDL, one gains several advantages. First, electronic components can be developed hierarchically. Second, VHDL includes time as an integral feature of the simulation. Third, conflicts between signals can be identified and resolved. Fourth, components can be perceived as simulating concurrently. Finally, the simulation can be validated against a VHDL testbench. These features combine to offer a flexible

simulation language that executes up to 2,000 events per second per MIPS (Coel:322).

2.3 Summary

To advance VHDL to the point where it can execute on a parallel computer, one must have a solid background in two areas. The first area is how discrete-event simulations, such as VHDL can be mapped to and executed on a parallel processor. This includes concerns like timing, rollback, message passing, and lookahead.

The second and more important area is VHDL itself. Unless the workings of sequential VHDL can be modelled, there is no hope of constructing a prototype for the parallel implementation of VHDL. As shown by Winters, VHDL not only provides a technical tool, but an economic one. By using VHDL, maintenance costs for chips being designed today will be lower. Since chip designs will only get larger and more complex, VHDL must grow in the same fashion. Chapter 3 provides a blueprint on how to parallelize VHDL.

III. Parallel Simulator Design

3.1 Introduction

For one to implement any design on a parallel computer, one must first comprehend how an equivalent algorithm works on a sequential computer. In this case, that is the sequential VHDL simulator. After sufficient study, one either designs a parallel simulator from the ground up or intercepts the process of a sequential simulator and attempts to switch the process onto a parallel course. Upon completion of the parallel simulator, one needs experiments to test theories about parallel VHDL simulation. This chapter describes the detailed layout of the above process as required by the AFIT environment.

3.2 An Example

Throughout this chapter, the reader is shown how to apply the theory discussed in Chapter 2 to parallel VHDL simulation. To better explain these ideas and link them together, an example of a full adder is used. The full adder example shows relationships between structural and behavioral architectures, timing considerations, and how to utilize multiple processors. Yet it is simple enough to be understood. Figure 3.1 provides the VHDL entity declaration, and Figure 3.2 depicts the corresponding circuit diagram.

The delay time shown in Figure 3.1 represents the longest propagation delay through the full adder. That means that if the user applied a '1' to X, Y, and CIN, then

9 ns later he should see a '1' for the value of both COUT and SUM. Therefore, the delay for the full adder, as a whole, is considered to be 9 ns.

```
Entity FULL_ADDER is
   Port (cin, x, y : in BIT := '0'; cout, sum : out BIT := '0');
   Constant Delay : Time := 9 ns;
end FULL_ADDER;
```

Figure 3.1 VHDL Code for Full Adder Entity.

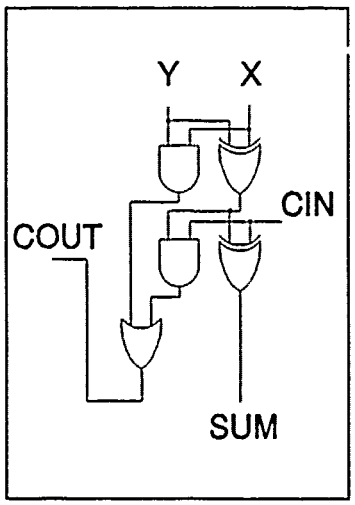

Figure 3.2 Full Adder Circuit Diagram.

The logic gates are the lowest level component that are modeled in the circuit. Therefore, each logic gate is described by a behavioral architecture. Figure 3.3 gives an example of the entity declaration and behavioral architecture for a simple AND gate (the OR and the XOR descriptions are similar). The full adder, on the other hand, can be structurally described as being made up of two half adders and an

```
Entity AND is
   Port (in_1, in_2 : in BIT := '0';
         out_1 : out BIT := '0');
   Constant Delay : Time := 3 ns;
end AND;

Architecture BEHAV_AND of AND is
begin
   OUT_1 <= IN_1 and IN_2 after delay;
end BEHAV_AND;
```

Figure 3.3 VHDL Entity and Behavioral Descriptions for an AND Gate.

```
Architecture STRUCT_FA of FULL_ADDER is
-- signal declarations
   signal cout_1, cout_2, sum_1 : BIT;
-- component declarations
   Component HALF_ADDER
      Port (x, y : in BIT; cout, sum : out BIT);
   end Component;
   Component OR_GATE
      Port (in_1, in_2 : in BIT; out_1 : out BIT);
   end Component;
begin
-- component instantiation
  HA1:  HALF_ADDER port map
        (x, y, cout_1, sum_1);
  HA2:  HALF_ADDER port map
        (sum_1, cin, cout_2, sum);
  OR1:  OR_GATE port map
        (cout_2, cout_1, cout);
end STRUCT_FA;
```

Figure 3.4 Structural Description of a Full Adder.

OR gate as shown by Figure 3.2 and described by the structural architecture in Figure 3.4 (a half adder is structurally described by an AND and an XOR).

Finally, Figure 3.5 shows the VHDL configuration of the full adder. It is the configuration file which tells the analyzer which architecture (behavioral or structural) is to be used for each component in the circuit.

```
use WORK.TEST_FULL_ADDER;
Configuration S_CONF_FA of TEST_FULL_ADDER is
   for INSTANTIATE_FULL_ADDER
      for FA : FULL_ADDER use
               Entity WORK.FULL_ADDER(STRUCT_FA);
         for STRUCT_FA
            for all : HALF_ADDER
               use Entity WORK.HALF_ADDER(STRUCT_HA);
                  for STRUCT_HA
                     for all : AND_GATE
                        use Entity WORK.AND_GATE(SIMPLE);
                     end for;
                     for all : XOR_GATE
                        use Entity WORK.XOR_GATE(SIMPLE);
                     end for;
                  end for;
            end for;
            for all : OR_GATE
               use Entity WORK.OR_GATE(SIMPLE);
            end for;
         end for;
      end for;
   end for;
end S_CONF_FA;
```

Figure 3.5 Configuration of a Full Adder.

3.3 Analysis of Sequential VHDL

3.3.1 Simulator Organization Figure 3.6 shows the typical organization of a standard sequential simulator used in support of the Standard VHDL 1076. It is made up of two major components: the analyzer and the simulator. The analyzer checks for syntactic and semantic errors. If there are none, it then produces an intermediate code

for later use by the simulator. The simulator (or kernel) takes the intermediate format and shapes it into executable code to execute the behaviors dictated by the original VHDL.

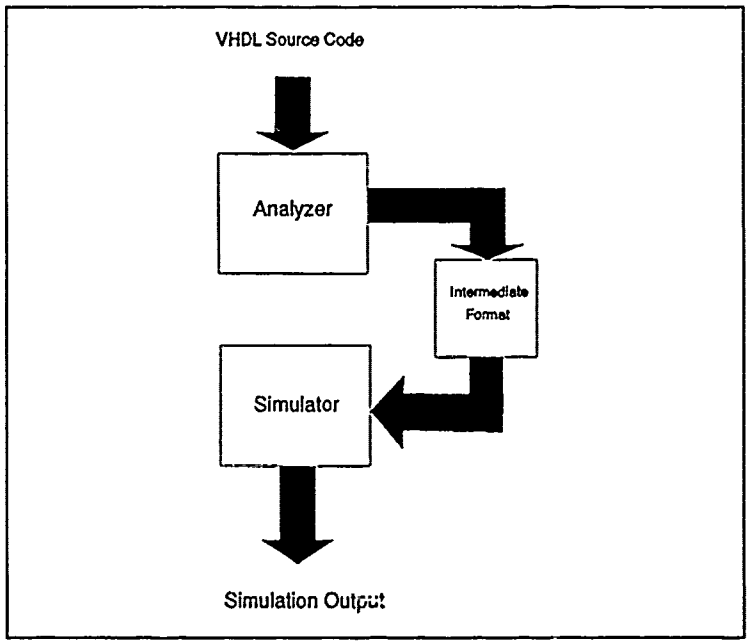

Figure 3.6 Simulator Organization.

When the circuit itself is simulated, the simulator alternates between simulating behaviors and updating signals (the connections between the behaviors being simulated). If one of the updated signals is an input to a behavior, that behavior is executed in the next simulation cycle to see if the new value of the input has any affect on the rest of the circuit. Thus, the simulator oscillates between executing behaviors and updating signals until reaching a quiescent state.

Assume, for example, that all signals (wires) in Figure 3.7 are initialized to '0' at time 0 ns. Assume also that the propagation delay through each gate is 3 ns. Next, a '1' is scheduled for CIN at 50 ns.

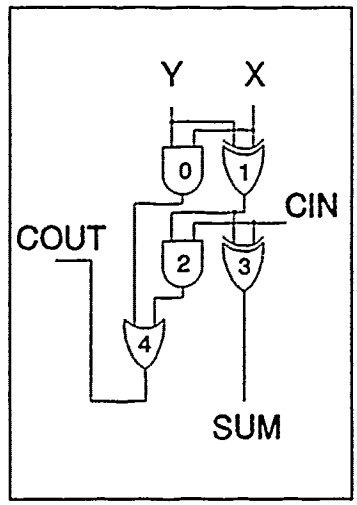

Figure 3.7 Full Adder Circuit Diagram.

The simulator first executes every gate at time 0. Following the paradigm, the next step is to update the signals. Since no signal values changed from the initialization step, the next event (signal update) takes place when CIN goes from '0' to '1' at time 50 ns. The simulator updates the simulation clock to 50 ns, and schedules for execution all behaviors (gates) for which CIN is an input. Next, the simulator executes the behavioral descriptions for gates 2 and 3, which results in scheduling a change to SUM from '0' to '1' at 53 ns. The output of gate 2 does not change. Then the simulator updates its clock to 53 ns and applies the signal change to SUM. Since SUM is not an input for any behavior and there are no more input changes for X, Y, or CIN, the system is quiescent at 53 ns and the simulation stops.

The heart of the simulator contains a core to keep track of the current simulation time and the signal update times. As the next signal update time arrives, the simulator updates the value for the signal and schedules the behavior for which the changing signal is an input. The output from that behavior is saved along with the time which the new

output will occur. When the simulation clock reaches that time, the simulator updates the signal, and schedules any behaviors for which the second signal is an input to execute. Thus, the cycle repeats until all signals are stable. The IEEE VHDL Language Reference Manual calls this cycle the "kernel process" (VLRM:B-8).

3.3.2 The Intermetrics VHDL Simulator Analyzing the Intermetrics VHDL simulator is easier at AFIT since the Air Force has the source code available to use as reference. Intermetrics design for their analyzer and simulator follows the pattern described above. Their system is made up of five stages: the analyzer, the model generator, the simulation builder, the simulation kernel, and the report generator.

The analyzer checks for syntactic and semantic errors and produces an intermediate format image of the VHDL source. The model generator produces C source files which takes intermediate IVAN files and produces from them C source files. The newly created C source is compiled into an object module, then the C source is deleted. The build phase links all of the new object modules as well as the Intermetrics VHDL simulator together and compiles them into a executable C module. The executable C module calls routines in the Intermetrics simulator which precludes one from running the simulation without the Intermetrics simulator. The kernel executes the simulation and the report generator produces a record of the signal changes for the user. A diagram of the Intermetrics process is provided in Figure 3.7.

With the assistance of personnel at Intermetrics, AFIT was provided with a switch which prevents the C source files from being deleted. This switch reports the name of the C source file that models the VHDL unit's behavior, its header file, and the corresponding

object file. Thus, one is able to analyze the C files in an attempt to see what constructs and variables in the VHDL source correspond to the constructs and variables in the C source code. After finding a correlation, one can manipulate the VHDL to alter the C source. This knowledge is crucial in that one cannot possibly understand what the C modules are doing unless he understands the VHDL behavior that is being simulated.

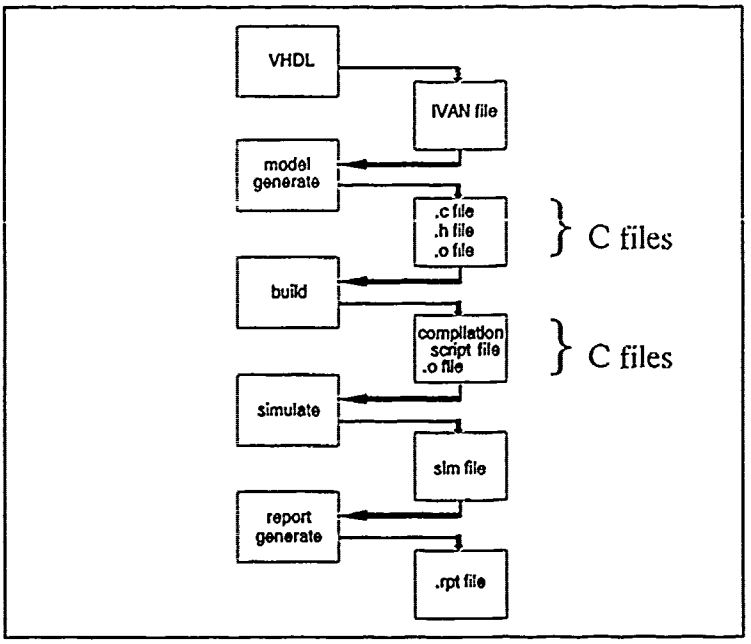

Figure 3.8 Intermetrics' Organization.

3.4 Approaches to Parallel VHDL Simulation

Now comes the task of creating a VHDL simulator to run on a parallel computer. One option is to write a VHDL simulator, ensuring total control of the process. However, writing a VHDL simulator from the ground up is either a multi-year effort or provides too limiting a subset of VHDL as to be useful. Therefore, rewriting the Intermetrics VHDL simulator to include commands for a parallel processor is a better option, since the Intermetrics source code is available to the Government. Thus, the plan for this thesis is

to intercept the process after the C source files are produced and modify the C source code for parallel execution.

3.5 Data Structures

After choosing to use the Intermetrics' analyzer and model generator as a preprocessor for parallel simulation, it is logical to implement the system using their data structures as well. There are four main data structures that are the key to the simulation. They are the data structures for signal records (sr), behavioral instances (bi), behavior list, and active records. Figure 3.10 shows the signal record structure and Table 3.1 gives the implementation of the full adder example (Figure 3.9).

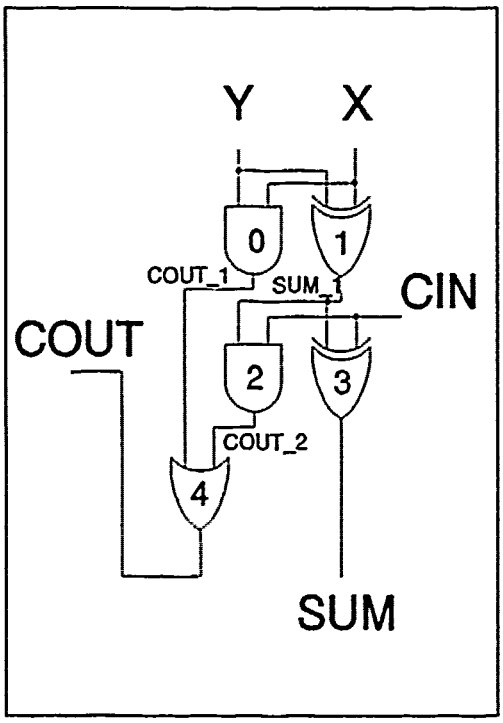

Figure 3.9 Full Adder Circuit Diagram.

id	= unique integer identifier
size	= number of bytes required for value
name	= pointer to the signal's name
cval	= integer offset for the signal's current value
conns	= pointer to a behavior list

Figure 3.10 Signal Record Data Structure.

The "id" field is a unique numeric identifier which corresponds to a particular signal (or wire) in the circuit being modeled. The "name" field is the name that is output whenever there is a change in the signal. Currently, since only digital 1's and 0's are used, the "size" field is always equal to one. All current signal values are kept in a common area in memory. The "cval" field is the offset from the beginning of the common area where this particular signal's current value may be found. Finally, "conns" is a pointer to a list of behavioral instances for which this signal is an input. Thus, when this signal's value changes, one uses the conns field to check if the change must be propagated through the circuit.

Table 3.1 Examples of Signal Records.

id	size	name	cval	conns
0	1	Y	0	0,1
1	1	X	1	0,1
2	1	CIN	2	2,3
3	1	COUT_1	3	4
4	1	SUM_1	4	2,3
5	1	COUT_2	5	4
6	1	SUM	6	
7	1	COUT	7	

```
id          = unique positive number
exec        = address in memory
input0      = pointer to input signal record
  :         :                    :
inputN      = pointer to input signal record
output0     = pointer to output signal record
  :         :                    :
outputN     = pointer to output signal record
```

Figure 3.11 Behavior Instance (BI) Data Structure.

As for the behavioral instances, Figure 3.11 shows their structure. The "id" is a unique identifier for each behavioral instance, and "exec" is the address of the behavioral instance subroutine. Pointing to the "exec" address and passing the pointer to the behavioral instance data structure as a parameter invokes the subroutine. Figure 3.12 displays part of the AND routine.

```
int AND_Output_Value;
  /* cv is the current value memory space address */
if (*((int *)(cv + Input0->cval))) {
  AND_Output_Value = *((int *)(cv + Input1->cval));
} else {
  AND_Output_Value = FALSE;
}
Gate_Delay = 3 ns;
posts8 (Output0, AND_Output_Value, Gate_Delay);
```

Figure 3.12 Code Excerpt From AND Routine.

The number of input and output signal record pointers that each behavioral instance (BI) has depends on the behavior being modeled. Table 3.2 shows the BIs at initialization for the full adder example. The structure of the behavior list is simply a

linked list of pointers to behavioral instances that are scheduled to execute during this simulation time. Figure 3.14 shows an example of the behavior list.

Table 3.2 Examples of Behavioral Instance Records.

id	exec	input0	input1	output0
0	address(AND)	0	1	3
1	address(XOR)	0	1	4
2	address(AND)	2	4	5
3	address(XOR)	2	4	6
4	address(OR)	3	5	7

```
time        = positive 32-bit integer
sr_ptr      = pointer to a signal record
value       = positive integer
next_sig_rec = pointer to an active record
```

Figure 3.13 Active Record Data Structure.

There is one final data structure which is important in understanding the VHDL simulator. That is the active record. Figure 3.13 depicts its structure. The "time" field contains the simulation time at which the event that the active record represents is to take place. The "sr_ptr" points to the signal record of the changing output signal. The "value" field of the active record holds the new value (logic '0' or '1'). Lastly, the "next_sig_rec" points to the next record in the active record list. Table 3.3 shows how CIN's (signal record 2's) change from '0' to '1' at 50 ns is represented on the active record list.

Table 3.3 Active Record Example.

time	sr_ptr	value	next_sig_rec
50	2 (CIN)	'1'	NULL

Figure 3.14 shows how the data structures shown above are interrelated for the full adder example. The relationships between these data structures are important in understanding how a basic discrete-event simulation works, as explained in the next section.

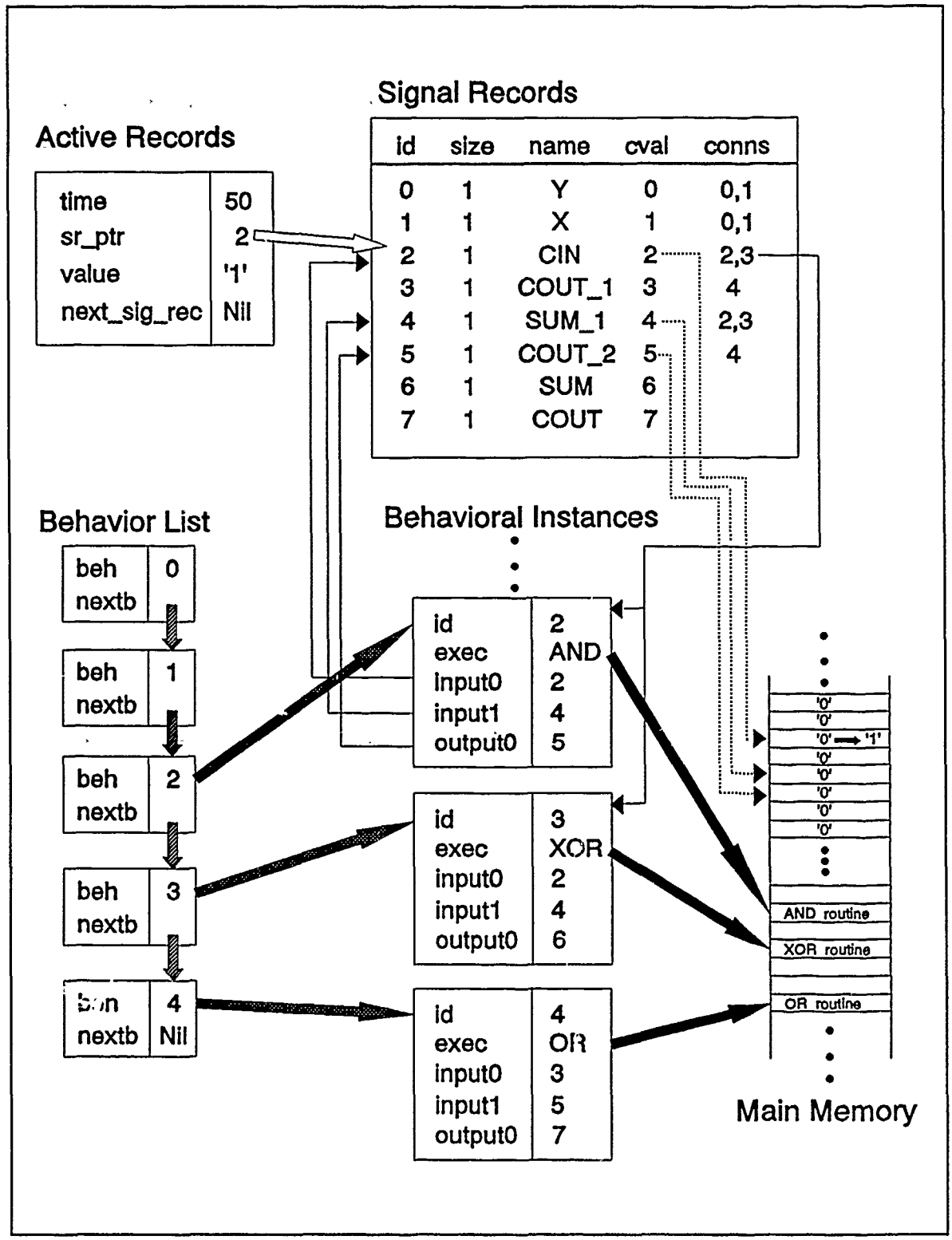

Figure 3.14 Interrelationship of VHDL Simulation Data Structures.

3.6 Process of Simulation

3.6.1 The Simulation Cycle Intermetrics' sequential simulator is the model for the parallel simulator. There are four routines which are needed to run the simulation. Figure 3.15 shows a diagram of the simulation cycle. The first is a post routine which posts each event (i.e., updates a signal's value) whenever a new output is generated.

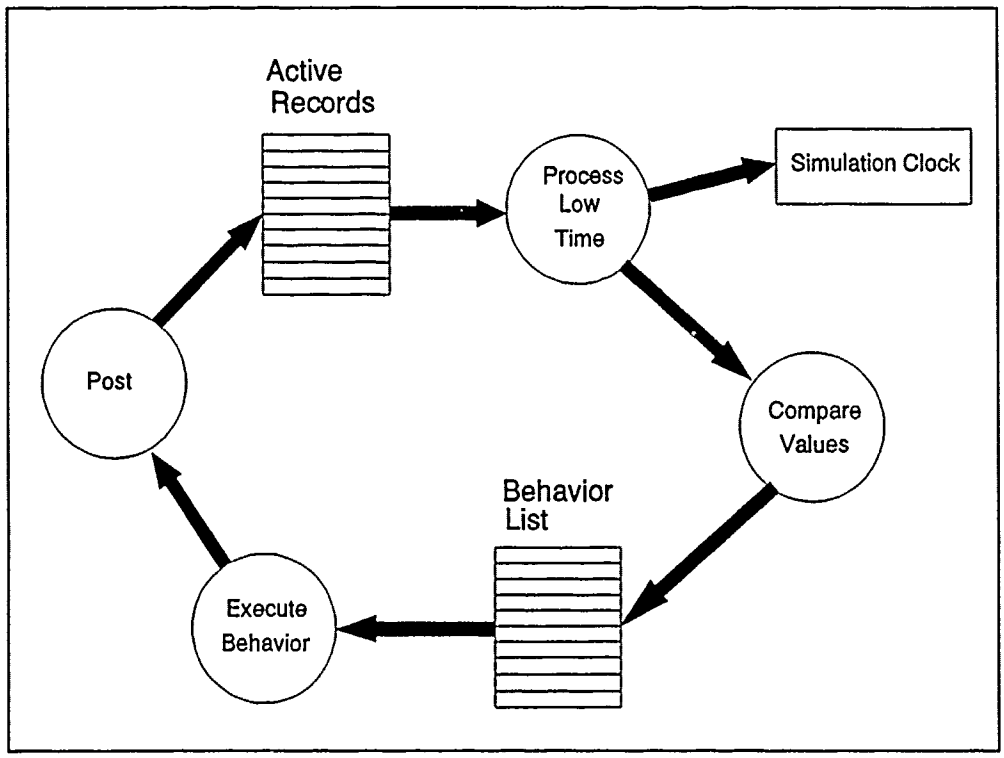

Figure 3.15 The VHDL Simulation Cycle.

The second is a process-low-time routine which analyzes the simulation times of each active record and advances its simulation clock to the lowest of those times. Then it removes any active records that are stamped with the new simulation time. If the signal's value in memory does not match the value on the active record, the computer assigns the value on the active record to the signal's value location. If, on the other hand, the signal's value in memory does match, the record is tossed away as if it had never

occurred. If the signal's value changes, any behavioral instances for which the changing signal is an input is added to the Behavior List. Table 3.4 shows the Behavior List.

Table 3.4 Behavior List.

RECORD NUMBER	beh	nextb
1	2	2
2	3	Nil

The last routine which is needed is one that removes each record on the Behavior List and executes the corresponding behavioral instance. The simulation, or execution, of the behavioral instance (e.g., AND, XOR, OR) calls the post routine to posts its output signals, as Figure 3.13 depicts. Thus, the cycle continues. Eventually, after all inputs have been exhausted, the simulation of the circuit reaches a quiescent state and the simulation ends.

Using the data structures presented in Figure 3.14, one can walk through the cycle in Figure 3.15. After initializing the simulation, the simulator adds all input changes to the Active Records using the post routine. Therefore, there is now one Active Record, as shown in Figure 3.14. Since there is only one entry in the Active Records, the low time is obviously 50 ns. The simulator updates the simulation clock to 50 ns and removes all Active Records which have a time stamp for 50 ns.

As the simulator removes the records, it compares the value on the record is compared with the current value for the signal pointed to by the "sr_ptr" field. Recall that all signals are initialized to '0'. Since the value on the Active Record is '1' the simulator

updates signal number 2 to a '1'. Looking at the Signal Record for signal number 2, the "conns" field shows that signal 2 is an input for both behavioral instance number 2 and behavioral instance number 3. Therefore, the simulator adds two records to the Behavior List, one for BI number 2 and one for BI number 3. Since there are no more Active Records to process for simulation time 50 ns, all signal updates are complete and executing the behavioral instances can begin.

The simulator executes BI number 2 by calling the AND routine and passing a pointer to the BI as a parameter for the AND routine. The AND routine pulls in the current values for "input0" and "input1", which correspond to "cv" plus "cval" for signal records 2 and 4. (Recall that "cv" is the address in memory where the current values of all of the signals are stored; "cval" is an offset from "cv.") The value for signal record 4 is still a '0', but signal record 2 has been updated to a '1' value. The AND routine calculates its output from the above inputs and posts this value by calling the Post routine. The AND routine passes its delay time, and the "output0" field so that the Post routine will know what signal to update (5, in this case) and when to perform the update. The Post routine then adds the simulation time to the delay time (50 + 3 = 53 ns) and puts a record to the Active Records with field values of "53, 5, '0'." The Post routine and the AND routine are now finished.

The Behavior List is not yet empty, so the simulator executes BI number 3. Its exec field points to the XOR routine, with input signal records 2 and 4. The simulator calculates the XOR output for the current values (at 50 ns) of signals 2 and 4, which are '1' and '0', respectively. This produces a '1' for an output value. The XOR gate also has a delay of 3 ns, so the output for signal 6 is posted for a '1' value, occurring at 53 ns (50

+ 3 = 53 ns). Thus, the simulator adds another Active Record with "53, 6, '1'" for the field values of the record. The Post and XOR routines are now finished. Table 3.5 shows what the Active Records look like at this point in the simulation.

Table 3.5 Current Active Records.

RECORD NUMBER	time	sr_ptr	value	next_sig_rec
1	53	5	'0'	2
2	53	6	'1'	NULL

So far, one cycle in the simulation (updated signals and executed behaviors) has completed. Continuing, the process-low-time routine looks for the lowest time among the Active Records, which is 53 ns. The simulator updates the simulation clock to this time and removes every Active Record with "53" in the time field. Removing record number 1, the simulator compares signal number 1's current value to the value on the active record. They are both '0' so the record is discarded. Active Record 2 points to Signal Record 6, which has a '0' for the current value. The active record has a '1' for the value, so the simulator changes the value for Signal Record 6 to a '1' and adds all Behavioral Instances to which Signal Record 6 is connected, to the Behavior List. Since Signal Record 6 has no entries in the "conns" field, it is not an input to any Behavioral Instance. Now both the Active Records and the Behavior List are empty, so the simulation is complete.

Figure 3.16 provides the pseudocode for the procedure described above. It goes into a little more detail since the pseudocode must check for all variations of the

simulation, but one should be able to correlate the example discussed in this section with Figure 3.16.

```
assign a numeric identifier to every signal
assign a numeric identifier to every behavioral instance (gate)
link behavioral instances to their input signal records
initialize all signal values for Simulation Time 0
schedule all behaviors to execute for time 0 by adding them
        to the Behavior List
while (Behavior List is not empty) loop
   while (Behavior List is not empty) loop
         execute the behavior
         post all signals output by adding it to the Active Record
               list with parameters (output signal record, current
               simulation time + delay time, new output value based on
               current value of inputs)
         delete behavior from behavior list
   end loop
   assign the lowest time on all of the Active Records to
               Simulation Time
   while (an Active Record has not been analyzed for this
               Simulation Time) loop
         if (Active Record Time matches Simulation Time)
               if (new output value = old output value)
                  delete record from Active Records    (throw it away)
               else
                  assign new output value to signal record
                  add any behavioral instance for which this signal is
                                 an input to the Behavior List
                  delete record from Active Records
               endif
               select next Active Record
         endif
   end loop
end loop
```

Figure 3.16 Pseudocode for Sequential VHDL Simulation

3.6.2 The VHDL Test Bench It is the test bench's VHDL source code which changes the input values for the circuit that is being simulated. A test bench is a structural

architecture, in and of itself, which uses the circuit that is being tested as its only subcomponent. In the case of the example, a test bench is used to test the full adder.

```
Entity TEST_FULL_ADDER is
end TEST_FULL_ADDER;

Architecture INSTANTIATE_FULL_ADDER of TEST_FULL_ADDER is
   Component FULL_ADDER
        port (cin, x, y : in BIT := '0'; cout, sum : out BIT := '0');
   end Component;
   Signal cin, x, y, cout, sum, cout_1, cout_2, sum_1, : BIT;
Begin
  FA : FULL_ADDER
    port map (cin, x, y, cout, sum);
  -- input test values into full adder
       cin <= '0' after 0 ns, '1' after 50 ns;
       x   <= '0' after 0 ns;
       y   <= '0' after 0 ns;
end INSTANTIATE_FULL_ADDER;
```

Figure 3.17 VHDL Source Code for the Full Adder Test Bench.

3.7 The iPSC/2 Hypercube

Intel's iPSC/2 Hypercube is a distributed memory, parallel processing computer which is made up of 2^n independent, 32-bit 80386 processors, where $n \geq 0$. Since there is no shared memory, the processors must convey information to each other via communication channels. These channels serve to connect the processors together along n dimensions (where n is the same value used above). Therefore, in order for a message to travel from one node to another, it may have to travel through n-1 interim nodes before finally reaching its destination.

The iPSC/2 incorporates a separate coprocessor, called a Direct Connect Module (DCM), to handle all message traffic passing through the node. Thus, the main CPU is free to concentrate on useful calculations. Since communication between nodes incurs a significant amount of overhead in terms of time, one must ensure that there is enough computation occurring on each node to warrant the communication time between itself and other nodes.

3.8 Parallel Simulation Issues

3.8.1 The Parallel Design The sequential model can now be extended to a parallel processor. The objective of parallel processing, to decompose a large problem into smaller problems, solve the small problems, then combine the results to solve the original larger problem. With this objective in mind, the obvious way to decompose a VHDL simulation is to simulate only a portion of the circuit on each processor. Therefore, individual gates are assigned to particular nodes.

As far as data structures are concerned, it is easier to initialize the entire data structure on each node than to partition it like the BIs. There are several reasons for this design decision. First, it avoids the overhead involved in analyzing which signals are attached to the BIs on the node. The position of cval offsets in memory on a node need not be determined since it is always the same. Organizing the generated C source code on a node to initialize only the signals and BIs on that node requires extensive modification for every new mapping. Because of these problems and others, the data structures on each node contain every signal and behavioral instance in the simulation.

However, a particular node is not concerned that the entire data structure is kept current. Only the signals which are inputs to a BI that assigned to the node or a system output signal (one that generates an output statement for the user) needs to be updated. Therefore, a node only updates and uses the portion of the data which it needs.

If a signal change takes place on node 0, and that signal is an input for a BI that resides on node 1, node 0 must somehow convey the signal change to node 1. Since in the sequential model, such a change is applied by posting an Active Record, the model can be extended and the change simply posted into node 1's Active Records. This is done by passing a message from node 0 to node 1 with the necessary information. In fact, if the signal is not an input for any BI on node 0, node 0 need not post the change to its own Active Records. These messages from node to node effectively link the simulation together to depict a coherent and correct VHDL simulation of the circuit. Figure 3.18 depicts two nodes posting messages to each other's Active Records.

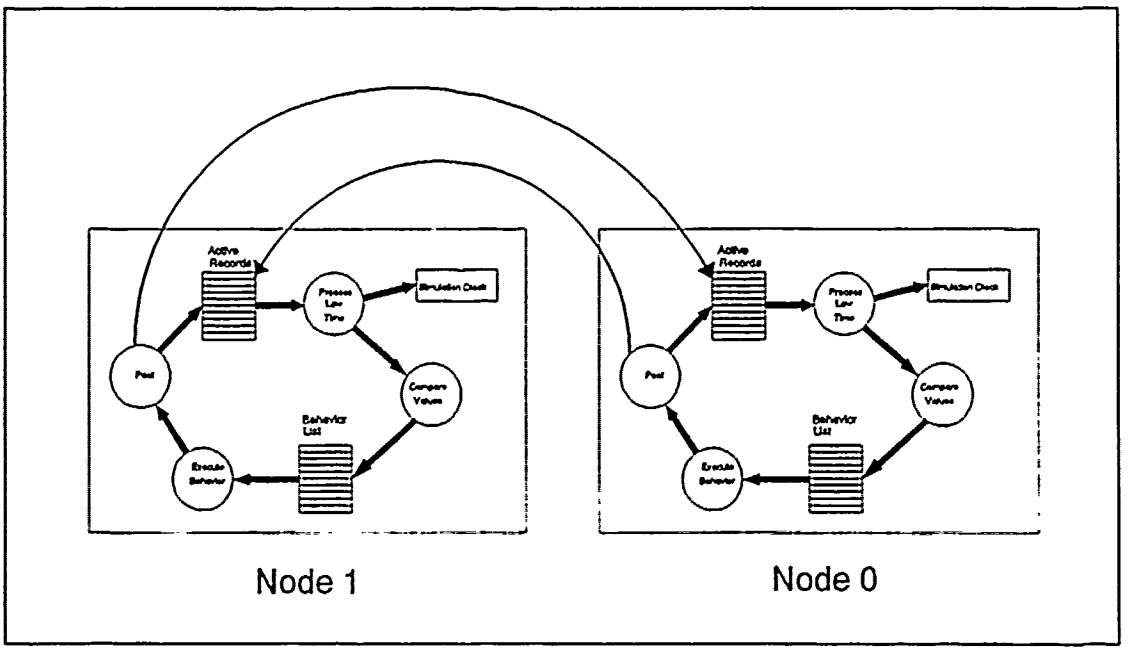

Figure 3.18 Simulation Model for Two Nodes.

To support the parallel model of Figure 3.18 the nodes must be aware of which behavioral instance is assigned to which node, which nodes are dependent on other nodes for signal update information, and when it is safe to execute the next event in the Active Records. The "queue.dat" file contains the node assignment of each behavioral instance in the simulation. The queue.dat file is made up of lines of tuples ("x, y") where x is the behavioral instance number and y is the node number to which x is assigned.

The "lp.arcs" file gives the node dependency. It also contains lines of tuples ("x, y") which represent "x passes messages to y." In such a case, x is called an upstream processor relative to y, and y is called a downstream processor relative to x. More is said about the queue.dat and lp.arcs files in Chapter 4.

The last problem is handled by using the Chandy-Misra paradigm. Any of the paradigms discussed in Chapter 2 might have been used. However, Chandy-Misra was picked for two reasons. First, Chandy-Misra is the paradigm used in most of the previous parallel work at AFIT; thus, this work is consistent with the current AFIT environment. Second, this thesis tests Fujimoto's and Subramanian's belief that VHDL simulations using Chandy-Misra are inherently limited in the amount of speedup they can realize. The next section covers how Chandy-Misra is implemented for parallel VHDL simulation.

3.8.2 Chandy-Misra Paradigm Every node is simulating its portion of the circuit as a logical process (LP). Thus, there exists an inherent "safe" time to which a downstream node might be able to process. That "safe" time for downstream nodes is the current simulation time of the upstream node, plus the minimum delay time of the upstream node's LP.

For example, suppose one LP is assigned per node and using two LPs are needed for the simulation. Further, assume that LP0's simulation clock is at 100 ns, LP1's clock is at 95 ns, and the last gate on LP0 has a delay of 5 ns. Since LP1 knows that LP0's clock is at 100 ns and that LP0's minimum delay time is 5 ns (its last gate delay), LP1 also knows that it cannot receive a signal change message from LP0 for any earlier than 105 ns (clock time + minimum delay time). Thus, it is safe for LP1 to simulate up to 105 ns. The key is to keep all downstream processes advised of updates to the simulation time.

Therefore, whenever the simulation clock changes on a node, the updated time plus the LP delay must be sent to all downstream nodes. This message announces to the downstream nodes the earliest simulation time at which the first node may cause an event to occur on the second node. Thus, each node can keep track of its "safe" time and simulate up to that time.

Further, according to Chandy-Misra, whenever a node receives a time update from an upstream node, it must add in its minimum propagation delay and pass the updated time onto all of its downstream nodes. This proves to be useful both in keeping the downstream processors busy and in preventing deadlock from occurring. Since there is no actual signal change attached to these messages, they are called "NULL" messages and although they provide no useful information other than simulation time, they are essential to the Chandy-Misra paradigm.

A problem can occur, however, when there is feedback between the LPs. To continue the example, assume now there is feedback from LP1 into LP0 and that LP1's minimum delay time is 5 ns. Now LP1 is dependent on LP0 and LP0 is dependent on LP1.

In continuing the simulation, LP0 must now wait for a message from LP1 since its safe time is 100 ns (LP1's time + LP1's minimum delay). LP1 simulates up to 105 ns, sends a message to LP0, and waits for a new safe time from LP0. It is now safe for LP0 to proceed. LP0 simulates up to 110 ns, sends a message to LP1, and waits for a new safe time from LP1. This "turn-taking" type of simulation will continue until the simulation ends. Since LP1 must wait while LP0 processes and vice-versa, any chance of parallel execution is eliminated. When the time required for communication between nodes is added to the time for turn-taking simulation, it takes longer to execute the simulation on two nodes than it takes to execute the simulation sequentially. Therefore, it is imperative to avoid feedback situations in parallel simulation.

3.8.3 Internode Dependence One interesting characteristic of parallel simulation is that an output signal on node 0 may also be an input signal on node 1. In such a case, the proper behavioral instance must be scheduled for the correct event time on node 1. This is the main reason for using the Chandy-Misra paradigm; to ensure that node 1 has not already processed beyond the new event time. Thus, a message must be sent from node 0 to node 1 which adds the active record into node 1's active record list.

If the signal is also an input for a behavior on any other node (including itself) the active record must be added to that node's active record list as well. Therefore, each node in the simulation is made aware if any such dependencies exist. Otherwise, a node must broadcast every signal change to every other node, most of which do not need the information. By using the lp.arcs file, sending and receiving irrelevant messages is avoided and more processing time is used for simulation.

The user places each Behavioral Instances on a particular node using the queue.dat file (exactly, how this is done is discussed in Chapter 4). Since the user know where every Behavioral Instance is and to what other Behavioral Instances a given BI is connected, he can map out the dependencies between the nodes. This dependency information is placed in the lp.arcs file.

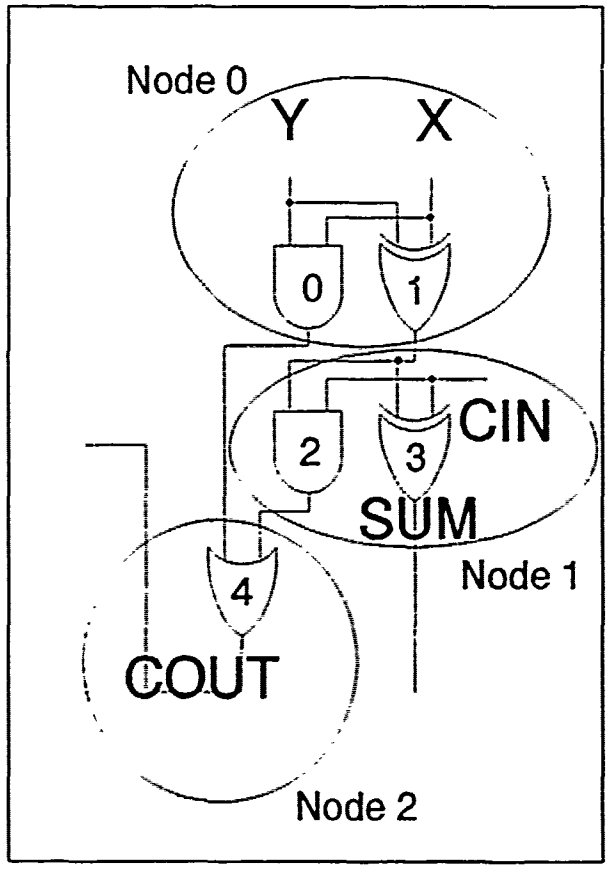

Figure 3.19 Full Adder Circuit Diagram.

Suppose gates 0 and 1 are mapped to node 0, gates 2 and 3 to node 1, and gate 4 to node 2, as shown in Figure 3.19. That means that node 2 is dependent on nodes 0 and 1 since the output of gates 0 and 2 are inputs for gate 4. Node 1 is dependent on

node 0 since gates 2 and 3 require the output of gate 1. The lp.arcs file for the above mapping is:

0 1
0 2
1 2

In the example above, LP2 is dependent upon both LP1 and LP0. The queue.dat file tells which behavioral instance is assigned to which node. The queue.dat file for this example is:

0 0
1 0
2 1
3 1
4 2

In the lp.arcs file, the second number shown on each line is dependent upon the first. For the queue.dat file, the first number is the behavioral instance identifier and the second number is the node on which the user wants the BI placed. The example above shows BI 2 (AND gate 2) and BI 3 (XOR gate 3) are assigned to node 1.

Since the mappings of the circuit varies depending on the number of nodes being used, the queue.dat and lp.arcs files change, as well. Changing these files every time the simulation is run on a different number of nodes is unacceptable. Therefore, the simulator uses a set of files called lpN.arcs and queueN.dat, where the "N" is equal to the number of nodes to be used.

3.8.4 Eliminating Unnecessary Features During the analysis of how to execute the simulation functions in parallel, some of the nonessential features of the sequential simulator were omitted in favor of design simplicity. For example, Intermetrics' simulator

includes a trace feature. The trace is used in debugging a circuit design in that it allows one to see what an internal signal's value may be at any point in the simulation.

Obviously, this feature requires a lot of overhead for state saving of every signal throughout the simulation. Since that aspect of the simulator is not a concern, all references to trace structures in the generated C programs are deleted. Choosing to leave out certain parts of the simulator increases the chances of running a complete parallel simulation without being distracted by unnecessary details. By completing a prototype simulator, efforts can be focused on the making parallel simulation feasible and efficient.

3.9 The Simulation

In order to test the parallel simulator, three structural models that go down to the gate level are used. The first model is a structural architecture of a ripple-carry adder. The ripple-carry adder is composed of eight full-adders. Each full-adder is made up of

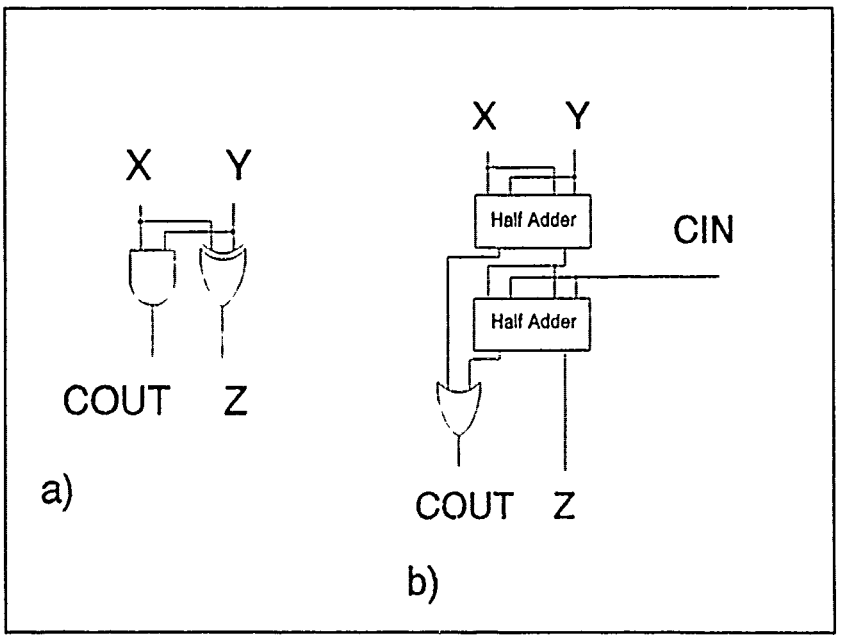

Figure 3.20 Configuration of a) Half Adder and b) Full Adder.

two half-adders and an OR gate. Each half-adder, in turn, is comprised of one AND gate and one XOR gate. The propagation delay through each of these gates is given in the VHDL source code as 3 nanoseconds (ns). Figures 3.20 and 3.21 show the complete component leveling from the half adder through the ripple-carry adder.

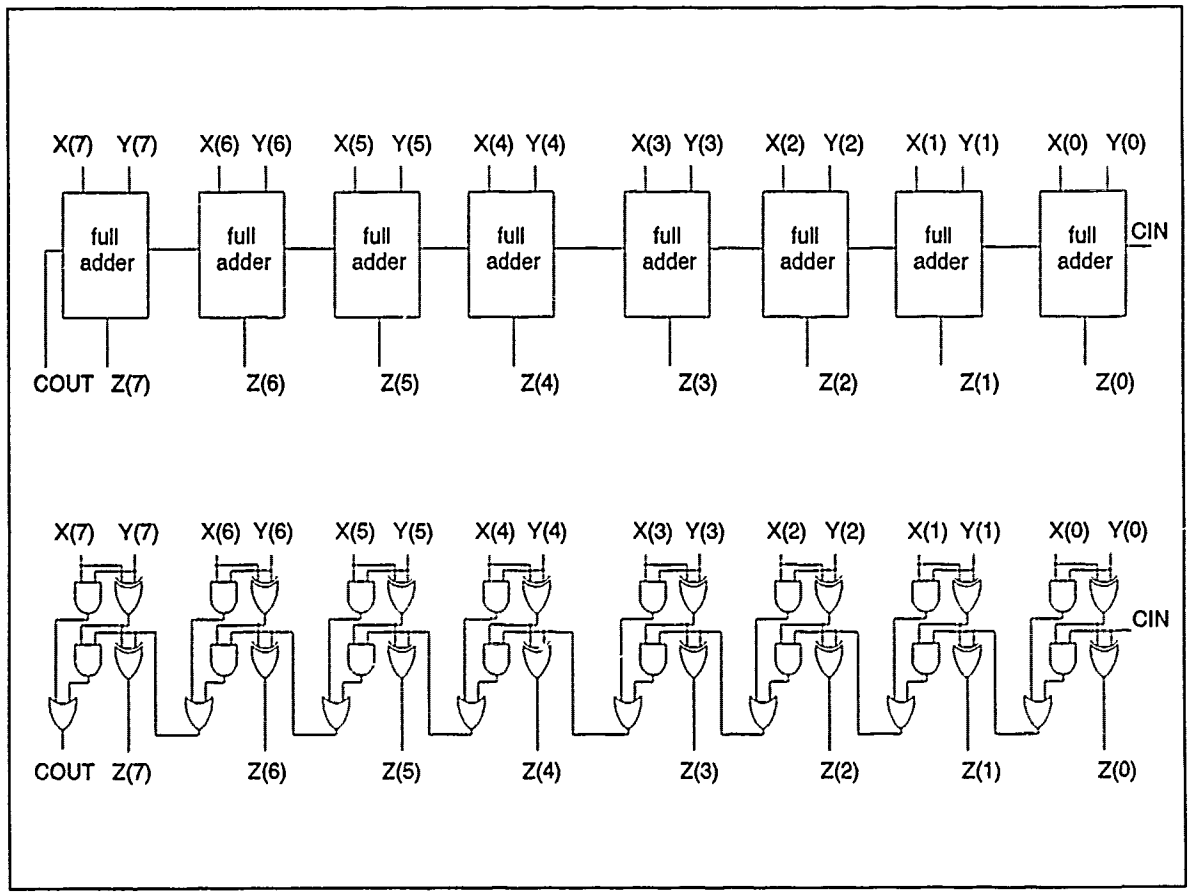

Figure 3.21 Ripple-Carry Adder Configuration.

Figures 3.22 and 3.23 display the circuit diagrams for the other two test cases, the Carry-Lookahead Adder and the Carry-Save Adder, respectively. The original VHDL source code and the corresponding Intermetrics-generated report is shown in Volume II. One may wish to refer to that volume for further information on the exact content of the VHDL source files.

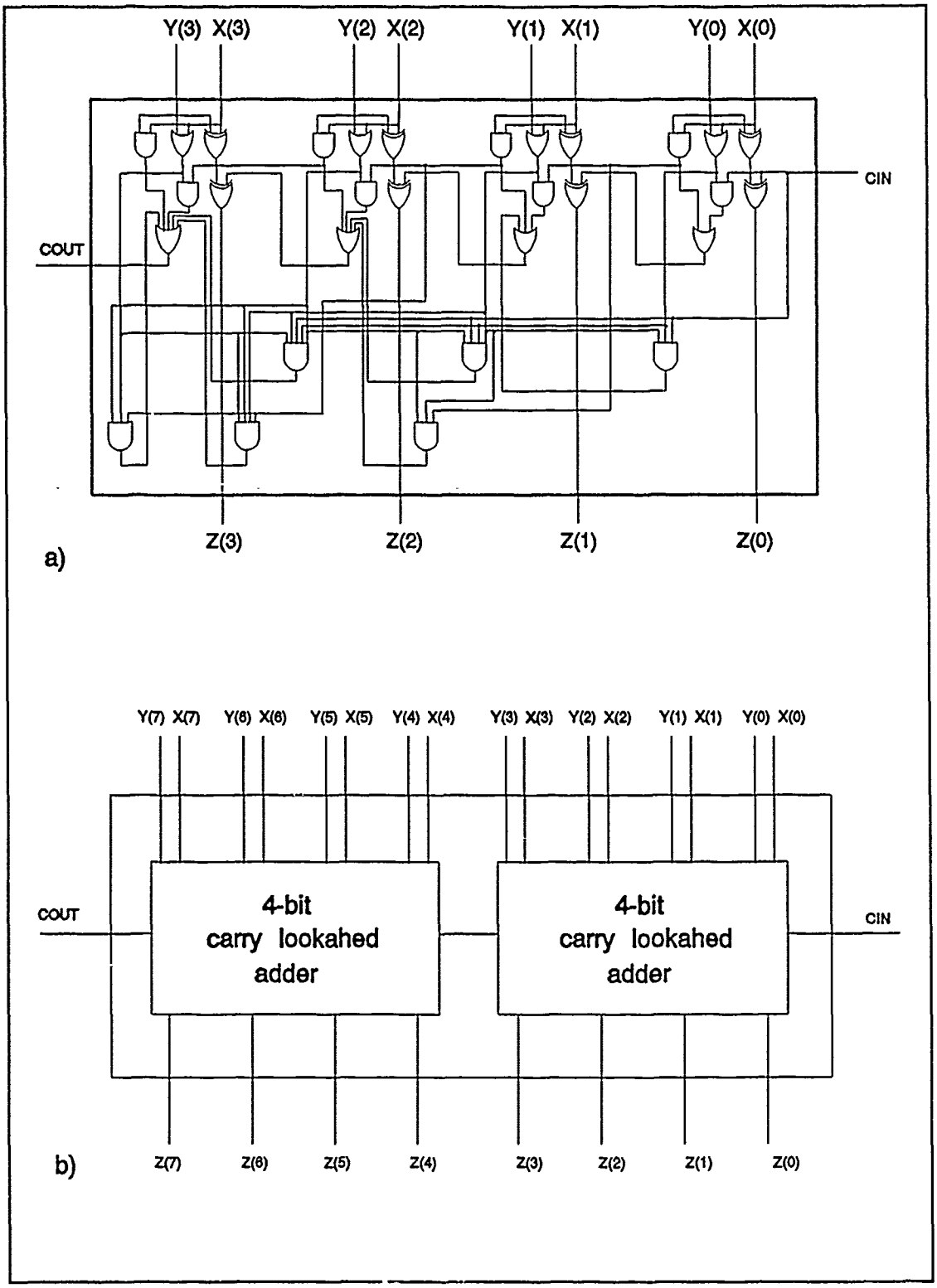

Figure 3.22 Carry-Lookahead Adder Configuration for a) 4 bits, and b) 8 bits.

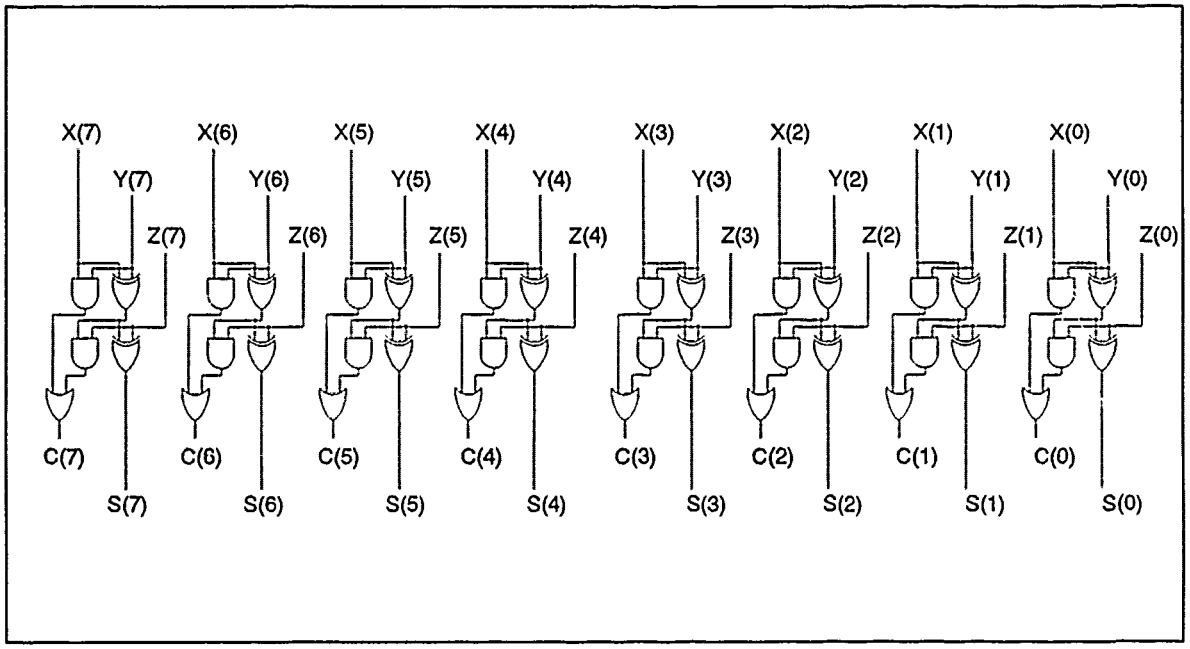

Figure 3.23 Carry-Save Adder Configuration.

3.10 Summary

Obviously, there are many factors to be considered before running VHDL simulations in parallel. One must have a model for sequential VHDL execution from which to start. After the sequential model is understood, a parallel architecture is chosen and any constraints which that architecture imposes are identified. Then the application is adapted for parallel execution, allowing for any hardware limitations, such as communication, mapping, and timing requirements. Once the parallel model is running, its output must be verified in that it matches what is produced by the sequential simulator. The actual implementation of the above issues is the subject of Chapter 4.

IV. Implementation

4.1 Introduction

This chapter explains the specific implementation of capturing the C source files generated by the VHDL compiler and integrating these files with hypercube subroutines to execute the simulation in parallel. With this objective in mind, this chapter explains how to implement the design specified in Chapter 3 on the iPSC/2 hypercube. By following the steps given here, several new, larger, and more interesting VHDL simulations can be run.

4.2 Editing Intermetrics-Generated C Source

4.2.1 Capturing C Source Code In a standard Intermetrics simulator, the "mg" or model generate command performs four functions in sequence. It extracts the model's behavior from the IVAN file which corresponds to the model the user wants generated. It creates C source and header files which will emulate the same behavior as in the IVAN description. Then the computer compiles the source file into an object file. Finally, the computer deletes the header and source files it just created.

By utilizing a switch (-debug=cknd) provided by Intermetrics, the C code that is generated by the model generate phase is not deleted. This ability allows the C code to be transferred to another computer, compiled there, and linked with parallel simulator routines. The advantage is that the code has already been syntactically and semantically checked by the Intermetrics' VHDL compiler, and it corresponds to the behavior that the user wishes to simulate. Figure 4.1 depicts a typical script file that may be used to compile and simulate

the full adder using Intermetrics' simulator. The commented "# mg" lines are not actually executed, but serve as reminders that these models must be compiled and generated before the full adder structural model is generated.

```
#!/bin/csh
vhdl fa_entity
vhdl fa_struct
vhdl test_full_adder
vhdl s_conf_fa
# mg '-debug=cknd or_gate(simple)'
# mg '-debug=cknd xor_gate(simple)'
# mg '-debug=cknd and_gate(simple)'
# mg '-debug=cknd half_adder(struct_ha)'
mg '-debug=cknd full_adder(struct_fa)'
mg '-debug=cknd test_full_adder(instantiate_full_adder)'
mg '-debug=cknd -top s_conf_fa'
build '-debug=cknd -replace -ker=struct_fa s_conf_fa'
sim struct_fa
rg struct_fa full_adder.rcl
```

Figure 4.1 Sample Compilation Script File.

Figure 4.2 displays a sample of a computer session in which the above model generate and build commands are entered. Notice that the model generator identifies the C source files which represent the behavior of the model being generated.

As shown in Figure 4.1 and 4.2, one may also use this "-debug=cknd" switch for the "build" phase. In the build phase, all of the models which make up the behavior of a circuit are linked together. Thus, by using the switch in building the simulation, the user can then access a file, called the "kernel com" file, which will tell him which source files need to be linked, where they are, and in what order they must be compiled.

```
USER COMMAND> mg '-debug=cknd full_adder(struct_fa)'
 Standard VHDL 1076 Support Environment Version 2.1b - 1 February 1990
 Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.
      Object_file : /usr/vhdl/shiplib/ron/FN2857.o
      H file      : /usr/vhdl/shiplib/ron/FN2858
      C file      : /usr/vhdl/shiplib/ron/FN2859.c
USER COMMAND> mg '-debug=cknd test_full_adder(instantiate_full_adder)'
 Standard VHDL 1076 Support Environment Version 2.1b - 1 February 1990
 Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.
      Object_file : /usr/vhdl/shiplib/ron/FN2867.o
      H file      : /usr/vhdl/shiplib/ron/FN2868
      C file      : /usr/vhdl/shiplib/ron/FN2869.c
USER COMMAND> mg '-debug=cknd -top s_conf_fa'
 Standard VHDL 1076 Support Environment Version 2.1b - 1 February 1990
 Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.
      Object_file : /usr/vhdl/shiplib/ron/FN2872.o
      H file      : /usr/vhdl/shiplib/ron/FN2873
      C file      : /usr/vhdl/shiplib/ron/FN2874.c
USER COMMAND> build '-debug=cknd -replace -ker=struct_fa s_conf_fa'
 Standard VHDL 1076 Support Environment Version 2.1b - 1 February 1990
 Copyright (C) 1990 Intermetrics, Inc.  All rights reserved.
      Kernel com file is /usr/vhdl/shiplib/ron/FN2877
```

Figure 4.2 Example User Session.

Figure 4.3 displays the build shell file, /usr/vhdl/shiplib/ron/FN2877, which is created by the Intermetrics' build phase. One can ignore the files in the "usr/vhdl/shiplib/std" subdirectory since they are Intermetrics' object files. One may either make note of the file numbers as the models are being generated, or instead add two to the ".o" file numbers given in the build shell file. (Notice that the C file number is always its corresponding object file number plus two.) The ".c" file in Figure 4.3 is the "main" routine which drives the program, so that one must link any other needed object modules while compiling this file.

```
#!/bin/csh
if ( $?VHDL_LIBSIM == 0 ) then
  if ( ! -e /usr/local/lib/libsim.a ) then
    echo NOLIB > bld_18042cnl.log
    exit 1
  endif
  setenv VHDL_LIBSIM -lsim
else if ( ! -e $VHDL_LIBSIM ) then
  echo LIBSM > bld_18042cnl.log
  exit 2
endif
cc -g -o /usr/vhdl/shiplib/ron/FN2875 /usr/vhdl/shiplib/ron/FN2879.c
/usr/vhdl/shiplib/ron/FN2872.o /usr/vhdl/shiplib/ron/FN230.o
/usr/vhdl/shiplib/ron/FN240.o /usr/vhdl/shiplib/ron/FN220.o
/usr/vhdl/shiplib/ron/FN2154.o /usr/vhdl/shiplib/ron/FN2857.o
/usr/vhdl/shiplib/ron/FN2867.o /usr/vhdl/shiplib/std/FN240.o
/usr/vhdl/shiplib/std/FN235.o /usr/vhdl/shiplib/std/FN225.o
/usr/vhdl/shiplib/std/FN25.o  $VHDL_LIBSIM -lcurses -ltermlib
-lm -lc >& bld_18042cnl.log
exit $status
```

Figure 4.3 Build Shell File.

The files above are each representative of some portion of the model to be simulated, as Table 4.1 shows. "FN2875" is the name of the executable file that all of the following files are linked into. It is used by the Intermetrics' simulator, and is not suitable for parallel simulation purposes; thus, it can be ignored. "FN2879.c" is the "main" C routine which simply calls six subroutines that actually run the simulation. "FN2872.o" is the object file for the configuration source file of the full adder, FN2874.c. Therefore, FN2874.c is copied into the big C source file. Files "FN230.o", "FN240.o", and "FN220.o" are the object files which represent the behavior models for an AND gate, XOR gate, and OR gate, respectively. So FN232.c, FN242.c, and FN222.c are appended to the big C source file. "FN2154.o" and "FN2857.o" are the object files of the structural models for a half adder and full adder,

respectively. Thus, FN2156.c and FN2859.c are appended. Finally, "FN2867.o" is the object file for the full adder test bench; so FN2869.c is appended.

Table 4.1 Intermetrics' Generated Files.

Object File	C File	Purpose of File
	FN2879.c	Main Program
FN2872.o	FN2874.c	Configuration
FN230.o	FN232.c	AND Model
FN240.o	FN242.c	XOR Model
FN220.o	FN222.c	OR Model
FN2154.o	FN2156.c	Half Adder Model
FN2857.o	FN2859.c	Full Adder Model
FN2867.o	FN2869.c	Test Bench

Notice, the order in which these files are concatenated is important. If a file is brought in out of order, errors result when the large C file is compiled. The order of the C files mentioned above would be FN2874.c followed by FN232.c, FN242.c, FN222.c, FN2156.c, FN2859.c, FN2869.c, and FN2879.c.

The C source files must be compiled and linked in the same order that is used by the standard VHDL simulator. Alternatively, one may opt to compile the files separately and link them at the end as is Intermetrics' approach. But a single file requires less modification. Therefore, this thesis only describes how to proceed after the C source has been concatenated into one large file.

4.2.2 Editing the Big C Source File Since the source file contains calls into the Intermetrics simulator routines, these routines must either be replaced by new routines or the calls must somehow be deleted. Most of the calls are to new routines for the parallel

simulator. However, some portions of the Intermetrics simulator, such as the trace routines, are too involved to replace. Therefore, the user must eliminate all references to the trace routine.

The only addition to the code is made to attach the correct name to each numeric signal identifier. Scalars and bit vectors must be handled differently, but neither proves to be a problem. The steps required, shown below, are currently performed by hand. However, this transformation process has potential to be easily automated. Figures 4.4 through 4.17 provide examples of how the steps below are implemented. The numbers that appear in braces at the end of some lines correspond to the step that applies to that line and are **not** part of the code.

1) The config file is brought in first, so it will "include" all necessary header files. Therefore, delete any line below the config file which contains the "# include" string.
2) Find all lines that contain "# include fn26" or "# include FN26" and delete them.
3) Change the paths of all remaining include files to the proper path. For example, if these header files are copied to the local directory, then "/usr/vhdl/shiplib/ron/FN2858" should be changed to "FN2858" as well.
4) Find all lines that contain "{trace" and delete from "trace" to the end of the line (leaving the "{").
5) Find all lines that contain "if (trceqp) {" and delete the **entire structure** associated with it (until the matching "}").
6) Delete any line containing the strings "trace" or "TRAREC" (if done by hand, "grep -v" works very well for this).
7) The last routine called by the main routine will be called "Z5xxxxxx" (where "xxxxxx" can be any series of numbers and letters). Insert "cv = init_cv ();" before the first line in the routine. The next line should be "Z1xxxxxx(NULL, NULL);." Add "sim_it (NULL);" as the third line.
8) There will be six subroutine calls in the "main" routine. The first will be "Z6xxxxxx ();". The sixth one will be Z5xxxxxx ();". The four in the middle will follow a "Z1xxxxxx ();" pattern. Delete each of the "Z1xxxxxx ();" calls.
9) Change the main routine name from "main" to "vhdl_main".

10) Find every occurrence of the "mksig" string. If the mksig function is assigned to a variable such as "(*cd).Zxxxxxx", it is a scalar, so one should proceed with step 10A for this occurrence. Otherwise, it represents a bit vector, therefore, proceed with step 10B.

10A) On the line below the mksig string, enter:

(*cd)(PARM1)->name = &(PARM2);

where PARM1 is the Zxxxxxx string to which mksig is being assigned. PARM2 is the first parameter that appears in the m_signal subroutine call that will be six lines below.

10B) Four lines above mksig, one can find:

lastsig = sigarr + NUM1 - NUM2;

where NUM1 and NUM2 are integer values. Below that line, enter:

loop_counter = NUM1 - NUM2;

where NUM1 and NUM2 are the same values as shown on the line above. This gives the index of the bit vector. Again find the "mksig" string.

Below that line, enter:

```
temp_name = (char*) calloc(sizeof(PARM1)+5, sizeof(char));
sprintf( temp_name, "%s(%d)", PARM1, loop_counter--);
(*(sigarr-1))->name = temp_name;
```

where PARM1 is a 22-character string which will appear 7 lines below as "Z30000xxx.xxxxxxxxxxxxx."

```
/* SIMPLE */
#include "simutl.h"                              [1]
#include "fn26"                                  [1,2]
#include "/usr/vhdl/shiplib/ron/FN221"           [1]
Z300008H_struct Z300008H =
```

Figure 4.4 Code Excerpt for Steps 1 and 2.

```
/* SIMPLE */
Z300008H_struct Z300008H =
```

Figure 4.5 Edited Code Excerpt for Steps 1 and 2.

```
/* S_CONF_FA */
#include "simutl.h"
#include "fn26"                                                    [2]
static char Z00000GO_trcbck []= {
 60, 60, 82, 79, 78, 62, 62, 83, 95, 67, 79, 78, 70, 95, 70, 65, 0 };
#include "/usr/vhdl/shiplib/ron/FN2868"                            [3]
#include "/usr/vhdl/shiplib/ron/FN231"                             [3]
```

Figure 4.6 Code Excerpt for Steps 2 and 3.

```
/* S_CONF_FA */
#include "simutl.h"
static char Z00000GO_trcbck []= {
 60, 60, 82, 79, 78, 62, 62, 83, 95, 67, 79, 78, 70, 95, 70, 65, 0 };
#include "FN2868"
#include "FN231"
```

Figure 4.7 Edited Code Excerpt for Steps 2 and 3.

```
{ register BOOL cond;
{trace_record->line_num =  18;                                     [4]
{
if (SQISNEG((&(*cd).Z00000CY_2592)))
            .
            .
            .
m_signal (Z30000SH.Z00000CY_3248,
(*cd).Z00000CY_3248->id,
Z4000001->Z0000001_11400);
if (trceqp) {                                                      [5]
   signame->name = Z30000SH.Z00000CY_3248;                         [5]
if (trace_signal_p(signame) == match) {                            [5]
(*cd).Z00000CY_3248->trp = TRUE;                                   [5]
}                                                                  [5]
}                                                                  [5]
trace_record->line_num =  9;                                       [6]
(*cd).Z00000CY_3320 = frms->Z00000CY_3320;
```

Figure 4.8 Code Excerpt for Steps 4, 5, and 6.

```
{ register BOOL cond;
{
{
if (SQISNEG((&(*cd).Z00000CY_2592)))
        .
        .
        .
m_signal (Z300008H.Z00000CY_3248,
(*cd).Z00000CY_3248->id,
Z4000001->Z0000001_11400);
(*cd).Z00000CY_3320 = frms->Z00000CY_3320;
```

Figure 4.9 Edited Code Excerpt for Steps 4, 5, and 6.

```
void
Z50000GO()                                  [7]
{
Z10000GO(NULL, NULL);                       [7]
timer();
close_sigdict();
if (!elaberror) close_sig_trace_file();     [6]
if (full) rptstats();
}
```

Figure 4.10 Code Excerpt for Steps 6 and 7.

```
void
Z50000GO()
{
cv = init_cv ();                            [7]
Z10000GO(NULL, NULL);
sim_it (NULL);                              [7]
timer();
close_sigdict();
if (full) rptstats();
}
```

Figure 4.11 Edited Code Excerpt for Steps 6 and 7.

```
main()                          [9]
{
  Z60000GO();
  Z1000001();                   [8]
  Z1000005();                   [8]
  Z1000007();                   [8]
  Z1000008();                   [8]
  Z50000GO();
}
```

Figure 4.12 Code Excerpt for Steps 8 and 9.

```
vhdl_main()
{
  Z60000GO();
  Z50000GO();
}
```

Figure 4.13 Edited Code Excerpt for Steps 8 and 9.

```
(*cd).Z00000EM_3368 = mksig( 1);              [10A]

(*cd).srcs->Z00000EM_3368 = NULL;



m_signal (Z30000EM.Z00000EM_3368,             [10A]
```

Figure 4.14 Code Excerpt for Step 10A.

```
(*cd).Z00000EM_3368 = mksig( 1);
(*cd)(Z00000EM_3368)->name = &(Z30000EM.Z00000EM_3368); [10A]
(*cd).srcs->Z00000EM_3368 = NULL;



m_signal (Z30000EM.Z00000EM_3368,
```

Figure 4.15 Edited Code Excerpt for Step 10A.

```
{ register SRPARR lastsig;

  lastsig = sigarr +  8 - 1;                    [10B]

  while (sigarr <= lastsig) {

*sigarr++ = mksig ( 1);                         [10B]

*srcarr++ = NULL;}

}

}

m_signal (Z30000EM.Z00000EM_4672,               [10B]
```

Figure 4.16 Code Excerpt for Step 10B.

```
{ register SRPARR lastsig;

  lastsig = sigarr +  8 - 1;
  loop_counter = 8 - 1;                                          [10B]
  while (sigarr <= lastsig) {

*sigarr++ = mksig ( 1);
temp_name = (char*) calloc(sizeof(Z30000EM.Z00000EM_4672)+5,
              sizeof(char));                                     [10B]
sprintf( temp_name, "%s(%d)", Z30000EM.Z00000EM_4672,
              loop_counter--);                                   [10B]
(*(sigarr-1))->name = temp_name;
*srcarr++ = NULL;}

}

}

m_signal (Z30000EM.Z00000EM_4672,
```

Figure 4.17 Edited Code Excerpt for Step 10B.

4.2.3 Compiling on the Hypercube Now that the file is ready to be transferred to the hypercube, one can use a file tranfer program (FTP) to upload it to the iPSC/2. Along with this C file, the user must also remember to transfer the "FN" header files that are included into the C file. The "simutl.h" file is already on the hypercube and differs from the simutl.h in the Intermetrics environment. Therefore, **do not** port this file.

Once all of the files are in place, one can begin compiling them. Figure 4.18 is an example of a make file for compiling the ripple-carry adder on the hypercube. One need only substitute their own circuit's file name for "full_adder" to use this make file for compilation and linking (assuming all of the files are together in the present directory).

```
#
# Makefile for creating/updating VHDL cube applications in C.
#
all:    host node full_adder.o pvsim.o
host:   host.o
        cc -o host host.o -host
node:   node.o full_adder.o pvsim.o
        cc -o node node.o full_adder.o pvsim.o -node
full_adder.o:  full_adder.c
        cc -c -w full_adder -g full_adder.c -full_adder
pvsim.o: pvsim.c
        cc -c -w pvsim -g pvsim.c -pvsim
clean:
        rm host node host.o node.o full_adder.o pvsim.o
```

Figure 4.18 Full Adder Makefile.

After the makefile has finished compiling and linking the code, one can execute the model. First, the desired cube size is allocated to the user by the user entering the appropriate "getcube" command. Then the user enters "host" to start the simulation run. Output

statements appear on the screen showing signal changes at discrete times. After the simulation is finished, the user releases the acquired cube with the "relcube" command. "RunitN" script files are provided to perform the getcube, host, and relcube steps, where "N" is equal to the desired cube size. An entry such as "runit8 >& out_file" redirects the output statements from an eight-node run to an output file with "out_file" as it name.

4.3 Simulation Subroutines

In order to get the Intermetrics-generated C source code to compile and run on the hypercube, one of two options has to be exercised for every subroutine call to the Intermetrics simulator. Either the call has to be deleted from the source code, or the subroutine has to be replaced with a new one in the parallel simulator. The first option works best for most of the trace calls since keeping the trace routines only adds overhead for a feature which is not yet needed. However, for most of the others it is more advanta-

Table 4.2 Parallel Simulation Routines.

1	init_cv
2	sim_it
3	update
4	get_event
5	mksig
6	strbi
7	setkck
8	init
9	padit
10	strsr
11	sqadd
12	posts8
13	rptast

geous to replace the subroutines with new ones. Of course there are also cases where altogether new routines are needed to perform the basic functions of discrete-event simulation. All of the routines that are used are described below and shown in Table 4.2. The source code is located in a single C file, "pvsim.c," which appears in Appendix A.

4.3.1 Subroutines From Sequential Models Some routines, such as the make-signal (mksig) routine, need to be done whether the simulator is sequential or parallel. Therefore, these tasks in parallel are implemented in the same manner as their sequential counterparts. The greatest advantage to utilizing sequential VHDL subroutines as a model is that it allows use of all of the data structures in the behavioral models. Creating these routines from scratch means significantly altering the model generated C source code to interface with any new data structures. Routines which are based on the sequential simulator are "mksig," "strbi," "setkck," and "init."

The "mksig" routine simply allocates space for a signal record and initializes it. "Strbi" puts behavioral instance records into the Behavior List for execution. "Setkck" attaches the signal records of inputs to a behavioral instance to that behavioral instance record. "Init" executes the behavioral instances before the simulation is started so that all signals will have a valid value associated with them.

4.3.2 Useless But Necessary Subroutines The next class of routines simply return to the calling program as soon as they are called. Since there is no productive work being done, it would speed up program execution by eliminating the need for these routines.

However, since editing the generated C source is not yet automated, a tradeoff must be made. Therefore, the time and effort required to find and extract the calls to these routines is traded for a slightly slower execution time. If one were to write a compiler for the C source code, it would then be feasible to take the calls, and hence the subroutines, out of the simulation environment. The subroutines are named "padit," "strsr," "sqadd," "m_signal," "rptstats," "read_input," "close_sigdict," "rptast," "timer," "rpterr," "pop," "tpop," "push," "m_real_type," "start_nonarray_comp," "sched," "rmtrrec," and "m_array_type."

4.3.3 Original Subroutines The last set of simulator subroutines are the ones that are required for the simulation to take place. The first is called "init_cv." Here, all system initialization takes place such as partitioning the circuit amongst the nodes in the hypercube, establishing message dependence between the nodes, and setting the simulation clock to zero.

The mapping that the user desires is put into the queue.dat file, where each behavioral instance is assigned to a particular node for the duration of the simulation. An example of this assignment process was shown in Section 3.8.3. Of course, how the user maps the circuit to the hypercube directly determines the dependence between the nodes. The user must put this dependence information into the lp.arcs file (also discussed in Section 3.8.3). It is the "init_cv" routine which reads in these files and places their information into array structures, which are accessed throughout the simulation.

The routine called "sim_it" is responsible for executing the behavioral instances. (Recall that the behavioral instances are the Intermetrics-generated models.) For example,

in simulating AND gate number three, the AND model is the behavioral instance which is executed with AND gate number three's record as a parameter. The AND routine will evaluate the values for the input signals attached to AND gate number three. If they are both '1' then its output will be '1' as well; otherwise, '0' is the output. The output signal is then posted by a call to the "posts8" routine. The calling routine, such as AND, sends the output signal's identifier, the AND gate's delay time, and the resultant output value, to the "post8" routine.

The "posts8" routine posts future events to the simulator by adding records to the Active Record list (as shown in Section 3.5.1). This routine determines which behavioral instances are affected by the potential signal change being posted. After the behavioral instances (BI) are marked, "posts8" locates the node on which the behavioral instance resides. If the BI is on the same node, "posts8" simply posts the signal change to its own Active Record list. If the BI resides on another node, "posts8" sends a message to that node and that node posts the signal change to its Active Record list. Since a signal can serve as an input to more than one BI (gate), there are times when the signal must be posted on the present node as well as sent to others. The "posts8" routine is also responsible for putting the correct signal change time on the Active Record. It accomplishes that by adding the current simulation time to the delay time of the calling behavioral instance.

"Update" is the where most of the simulation control code resides. This routine is responsible for finding the next time for the simulation clock. In a sequential simulation, it accomplishes the task by simply taking the lowest time on the Active Record list as the new simulation time. In parallel simulation, however, each node must be assured that it will not

receive a message from any other node for a signal change in the past. Any given node identifies which nodes might send it messages through the lp.arcs and the queue.dat files.

"Update" compares the lowest next event time on the Active Record list with the safe times for every other upstream node (upstream meaning one that might send a message). If it is lower, then the simulation clock is updated, Active Records for the new simulation time are analyzed, and those that qualify are put on the behavior list. If, however, the next event time on the Active Record list is greater than the safe times for any of the nodes, then the node must wait for messages from other nodes, so it calls the "get_event" routine.

"Get_event" processes all incoming messages, updating safe times of upstream nodes whenever it receives a NULL message. It also adds a signal change to the Active Records when it receives a signal change message. This message-processing action continues until it is safe to proceed with the lowest next event time in the Active Record List.

Note that a node may receive a signal change message that occurs earlier in the simulation than the current next event. This is precisely why a node is not allowed to proceed past the lowest safe time. If such a signal change message is received, the message is added to the Active Record list, the new low time is set to the new message's event time, and the safe times are reevaluated. Eventually, the node receives messages which update all of the safe times past the next event time; thus, deadlock is prevented.

Once the simulation time is safe, all of the Active Record signal values tagged with that time are evaluated. If the new value equals the old value then the record can be thrown away without any further processing. This is because propagating a signal value that has not changed would cause other behavioral instances to execute, with no consequent

change to their output values, and so on. Thus, VHDL simulations simply consume signal values that do not change, preventing a lot of useless processing. If there is a change to the signal value, then the simulator updates the signal value in memory, prints an output statement to document the signal transition, and assesses the next candidate Active Record signal.

Once all of the signals are updated, the simulator executes the behavioral instances. This is done by going through the Active Records for the current simulation time again. Records that match the simulation time are the signal values that are changing and each contains the addresses of the behavioral instances which it stimulates. If the behavioral instance is on the current node, the processor schedules it for execution by placing it on the Behavior List. Recall that messages are sent out to other nodes, if necessary, in the posts8 routine. Therefore, a signal change that affects behavioral instances on downstream nodes has already been accounted for.

Once the Behavior List for the current simulation time is complete, the list is ready for execution. Control is given back to the sim_it routine and the simulation continues as described above. Figure 4.19 is a form of psuedocode that recapitulates what was described above (i.e., the parallel simulation process) for each node. One can use the full adder example in Appendix B to illustrate the parallel algorithm shown in Figure 4.19.

```
1     assign a numeric identifier to every signal
2     assign a numeric identifier to every behavioral instance (gate)
            and assign it to a node as in the queue.dat file
3     if (my node equals y in a pair (x,y) listed in the lp.arcs file) then
4           assign MAX_TIME to safe_time[y] to show what nodes will send
                  messages to my node
5           assign FALSE to waiting[x]
6     endif
7     if (my node equals x in a pair (x,y) listed in the lp.arcs file) then
            assign MIN_DELAY to safe_time[y]
8           assign TRUE to waiting[y] to identify downstream nodes
9     endif
10    link behavioral instances to their input signal records
11    initialize all signal values for Simulation Time 0
12    schedule all behaviors on my node to execute for time 0
            by adding them to the Behavior List
13    while (Behavior List or Active Records are not empty
             or a safe_time is less than MAX_TIME) loop
14          while (Behavior List is not empty) loop
15                execute the behavior
16                if (the resultant output signal connects with a behavioral
                              instance that resides on my node) then
17                      post the signal by adding it to my Active Record list with
                              parameters (output signal record, current simula-
                              tion time + delay time, new output value based
                              on current value of inputs)
18                endif
19                if (the resultant output signal connects with a behavioral
                              instance on another node) then
20                      send a message to that node with the parameters (my
                              node number, output signal record, current
                              simulation time + delay time, new output value
                              based on current value of inputs)
21                endif
22                delete behavior from behavior list
23          end loop
24          assign the lowest time on all of the Active Records to Low_Time
25          while (a safe_time is less than or equal to Low_Time) loop
26                receive a message
27                if (it is a NULL message) then
28                      update the safe_time for the node that it is from
```

Figure 4.19 Parallel Simulation Pseudocode.

```
29                  else
30                      add the record to the Active Record list
31                      reevaluate Low_Time
32                  endif
33              end loop
34              assign Low_Time to Simulation_Time
35              send the new Simulation_Time to all nodes that are waiting[]
36              while (an Active Record has not been analyzed for this
                        Simulation Time) loop
37                  if (Active Record Time matches Simulation Time) then
38                      if (new output value = old output value) then
39                          delete record from Active Records (throw it away)
40                      else
41                          assign new output value to signal record
42                          add any behavioral instance for which this signal
                                    is an input to the Behavior List
43                          delete record from Active Records
44                      endif
45                  endif
46                  go to next Active Record
47              end loop
48      end loop
49      send an "All Done" message to all nodes that are waiting[]
```

Figure 4.19 (continued) Parallel Simulation Pseudocode.

4.4 Concerns for Parallel Simulation of VHDL

The main issue in a parallel discrete-event simulation is which timing protocol to use. This research uses a variant on the Chandy-Misra paradigm. Chandy-Misra dictates that a message will be sent to downstream nodes whenever one of two events occurs. The first is whenever the simulation time changes so that a downstream node will not be waiting unnecessarily. The second is when the lowest safe_time changes on a node, the safe time should be added to the delay time, and that too should be passed along to downstream nodes. The second rule serves to speed up the simulation if a node has very few events, and

those are separated by relatively long intervals of simulation time.

The first Chandy-Misra guideline is adhered to for the parallel simulator in that a NULL message is sent whenever there is a simulation time change. Since the simulation time can never be greater than the lowest safe time, rule number two is upheld as well. Nonetheless, one may find greater overall speedup for certain circuits if some of the restrictions are relaxed. For example, since the simulation time of some circuits changes frequently and in small increments, one may find that most of the NULL messages are somewhat redundant; providing little advancement for the costly overhead of sending and receiving a message.

In conventional simulations, a process can simulate up to and including events that occur at the safe time. However, the nature of a VHDL simulation is such that a node can only simulate events up to the lowest safe time. That is because two steps are taken in the simulation: 1) the signals for that time are updated; and 2) the behavioral instances are executed. If the second step is taken, one may be executing a behavioral instance which uses input data from another node. That input signal may have changed at the same simulation time, but the downstream node has not yet received the message. The resultant output would therefore be in error. Figure 4.20 shows a portion of a simulated circuit which will demonstrate this phenomenon.

Suppose the simulation time on node 1 is 95 ns. The current value is '1' for signal B, and the next event on the Active Records list is for signal A to go from '0' to '1' at time 100 ns. In accordance with Chandy-Misra, node 0 has sent a NULL message to notify node 1 that no messages will be sent earlier than 100 ns (simulation time + gate delay); hence,

safe_time[0] equals 100. Node 1 calculates its Low_Time to be 100 ns, so it updates its Simulation_Time to 100 ns.

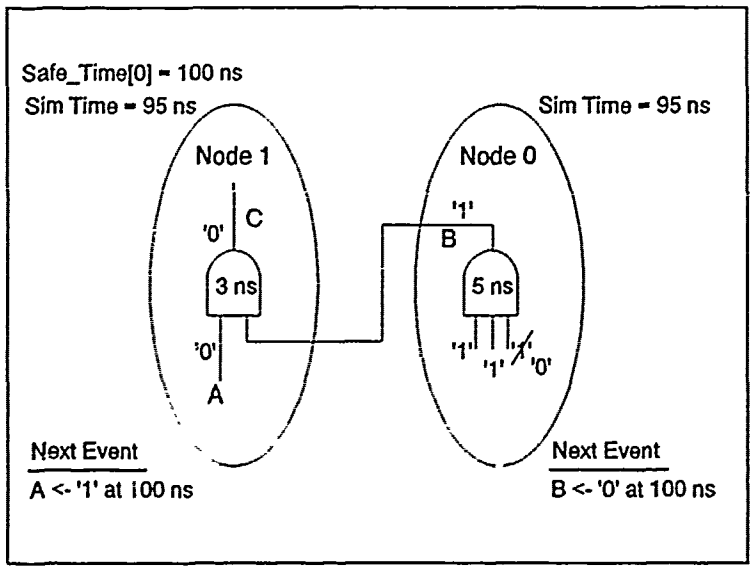

Figure 4.20 Example Circuit for Parallel Simulation.

If conventional simulation guidelines are followed, a node can simulate any event that occurs at the Simulation_Time. So the value of signal A is changed to '1' and the AND gate is scheduled to simulate. Since the Active Records list is now empty, the AND gate is executed. Its current inputs are both '1' so the resultant output, C, is calculated to be a '1' at 103 ns (assuming a gate delay of 3 ns). The output signal change is posted into the Active Record list.

The new Low_Time is 103 ns, but safe_time[0] is still 100 ns, so node 1 must wait for a message from node 0 to update the safe_time before proceeding. Instead of sending a NULL message, node 0 sends a signal change message for signal B going from '1' to '0' at 100 ns. 100 ns becomes the new Low_Time, signal B is updated to '0', and the AND gate

is executed again. Signal C is posted for the value '0' and the time 103 ns onto the Active Records list.

When the Simulation_Time on node 1 reaches 103 ns, the simulator prints that signal C is changing from '0' to '1' at 103 ns, followed by another line that shows signal C is changing from '1' to '0' at 103 ns. Any gates that use signal C for input are now simulated using the correct value of '0' for signal C and the simulation continues correctly. Since the initial change to C was calculated with incorrect inputs at time 100 ns, the simulation records an errant signal change for C. That one print statement is the only flag the user will see that indicates a bug in the simulation.

To avoid this unfortunate series of circumstances, one solution is only to simulate up to the lowest safe_time. This guarantees, as is Chandy and Misra's intention, that a node does not process an event using invalid data. An alternative solution is to scan the Active Records list before any signals are updated to see if there are two occurrences of any signal change which conflict with each other. However, due to the way that Intermetrics inserts and removes the Active Records, which is the model for this research, the records are not ordered as they are inserted. Thus, for a list of length n, the node needs to traverse the list twice every time the simulation time is updated. This is at least equal to the number of discrete-event times as we have on the node -- this overhead is unacceptable. Therefore, it is a significantly better solution to sit idle and wait for a message which will increment the safe_time.

Perhaps, in the future, Intermetrics' model can be put aside and the data structures handled a little more efficiently. In running large, complex simulations in parallel, as is the

goal; efficiency is a priority. For right now, however, the model does work -- perhaps not as well as it might -- but it does work.

4.5 Mapping to the iPSC/2

Circuit partitioning must be flexible so that different mappings may be tried for various sizes of hypercubes. Utilizing the "queueN.dat" files (where "N" is equal to the desired number of nodes) as described in Section 3.8.3, the user is able to set up and save the best-performance mapping for each cube size. Note that no recompilation is necessary -- the simulator reads in the appropriate queue.dat file during the simulation run and maps the behavioral instances appropriately.

In order to perform any type of mapping using the queue.dat file, one must first know how many behavioral instances there are and which is connected to which. This is set up within the Intermetrics' generated C code. As such, it is not readily available to the user.

To make retrieving this information easier, a switch within pvsim.c is set which prints out information for mapping the circuit. The switch is called "MAPPING." Set both this value and the value for "OUTPUT" to "1", recompile pvsim.c, and run the simulation on one node. Along with the regular output statements, the user sees when a behavioral instance is executing and the subsequent behavioral instances the executing BI affects. Figure 4.21 shows an example.

Figure 4.21's output shows that behavioral instance 0's output is an input for behavioral instance 4. Likewise, behavioral instance 1's output signal is an input signal for behavioral instances 2 and 3. Since there is no BI posted after behavioral instance 3

executes, that indicates that BI 3's output is a system output. Now that the dependencies between BIs is known, mapping the circuit is easier.

```
0->Executing behavior number 0.
0-> Add behav 4 to my list (node 0) for time 3000000
0->Executing behavior number 1.
0-> Add behav 2 to my list (node 0) for time 3000000
0-> Add behav 3 to my list (node 0) for time 3000000
0->Executing behavior number 2.
0-> Add behav 4 to my list (node 0) for time 3000000
0->Executing behavior number 3.
0->Executing behavior number 4.
```

Figure 4.21 MAPPING Output Example.

The approach used for mapping an application to the hypercube is to map it to eight nodes first (the largest cube size available at AFIT). Using trial-and-error, one can identify the mapping which gives the fastest simulation run time. The user translates the mapping to four nodes by combining what was mapped on every two nodes onto a single node. For example, BIs mapped to nodes 0 and 1 for the 8-node run are mapped to node 0 for the 4-node run. BIs mapped to nodes 2 and 3 are mapped to node 1 for the 4-node run, and so on.

Scaling a simulation up to a larger cube size is not as easy. It is clear that one should allocate the behavioral instances on a single node to two nodes on the larger cube. But there is no simple formula for identifying which behavioral instances to map to which node. The user can only bear in mind general rules-of-thumb, such as avoiding unnecessary communication between nodes, balancing the computation that occurs on each node, and avoiding feedback loops where possible.

4.6 Summary

This chapter has shown how to implement the design given in Chapter 3. The full adder example in Appendix B takes the reader through each step in this chapter in detail. The reader must be concerned with issues such as modeling the sequential behavior, how to map that behavior for parallel execution, timing and synchronization paradigms, and a host of others. Once these issues are worked out, simulation of larger circuits is simply a matter of expanding the experience base. The next chapter introduces test cases that are simply expansions of the full adder example used up to this point.

V. Test Cases

5.1 Introduction

The test cases presented in this chapter were chosen based on three criteria. First, they each are relatively simple to simulate in sequential VHDL and one starts with simple models when testing new concepts. Second, although simple, they each are complex enough to have a structural design which goes down to the gate level; thus, a structural architecture can be parallelized. Finally, each provides a different number of behavioral instances for simulation. The following sections point out each test case's redeeming qualities in detail.

5.2 The Carry-Save Adder

As mentioned in the previous chapter, when mapping an application to the iPSC/2 hypercube, one tries to maximize the computation to communication ratio on each node. Finding the rare application in which the communication costs are zero means finding the perfect application for the hypercube. Such is the case with the carry-save adder.

Because the carry-save adders is a simple building block for larger circuits such as multipliers, it is of little interest in and of itself. For the purpose of this research, however, it not only provides an excellent test case for the hypercube, but it is also the first step to creating much larger VHDL simulations for parallel execution.

As shown in Figure 3.23, a carry-save adder is made up of eight separate and independent full adders. Since there are no signal connections between the full adders, there is no communication needed between the nodes. Consequently, communication time is zero and the computation to communication ratio is maximized.

5.3 The Ripple-Carry Adder

The ripple-carry adder, shown in Figure 3.21 is similar to the carry-save adder in that it also is made up of eight full adders. Likewise, the circuit is partitioned in the same manner, putting 8/p full adders per node for p nodes. But the ripple-carry adder propagates its carry from one full adder to the next. This signal propagation means communication is necessary between the nodes. Therefore, the ripple-carry adder is not expected to gain as much speedup as the number of nodes are increased, as the carry-save adder.

5.4 The Carry-Lookahead Adder

The most intriguing circuit to partition is the carry-lookahead circuit, shown in Figure 3.22. This is due not only to the increased number of behavioral instances (76 versus 60), but also to the difference in delay times. Standard two-input gates are set to a delay of 3 ns. But the carry-lookahead adder contains three-, four-, and five-input AND and OR gates, which have a delay of 5 ns. Therefore, the timing throughout the circuit is much more complex and dynamic.

The carry-lookahead adder was initially partitioned as were the other two adders, by allocating a full adder to each node (as shown in Figure 5.1). After simulating this circuit several times, a noticable discrepancy between the completion times for the first and last nodes is evident. Table 5.1 shows one such set of execution times (in milliseconds) for the mapping of Figure 5.1.

By partitioning the circuit by trial and error, one can often find a better way to balance the workload. Since it is behavioral instances that are moved to other processors,

and since each behavioral instance is representative of a gate in the circuit, one can use the circuit diagram to reallocate gates to processors. As Section 3.8.2 pointed out, a constraint which must be remembered is to avoid unnecessary feedback between nodes. Such feedback

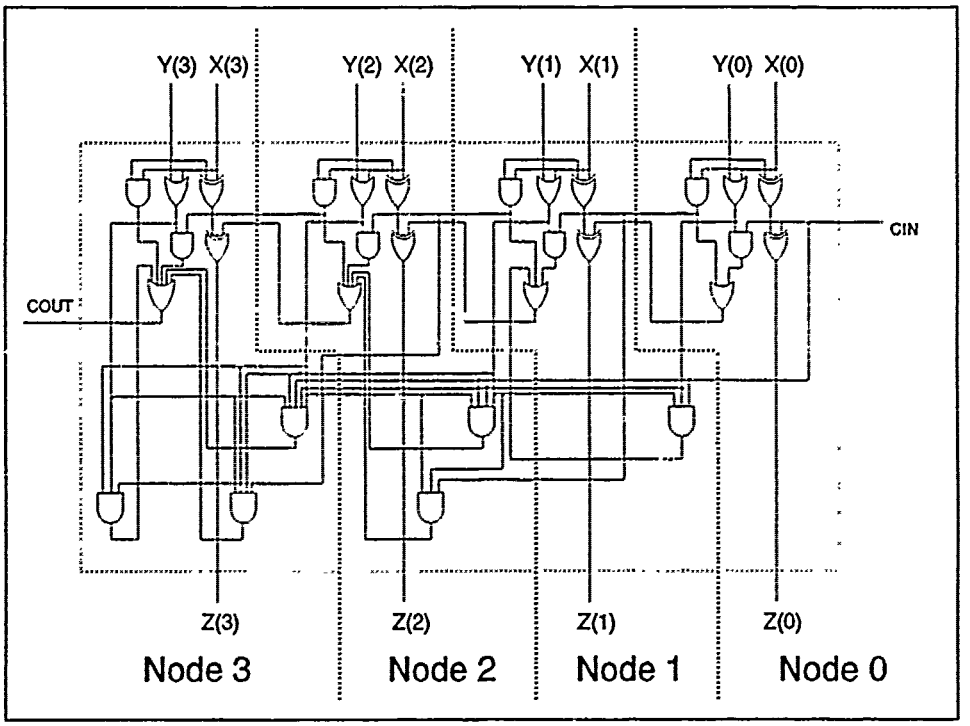

Figure 5.1 Approach 1 for Carry-Lookahead Partitioning.

Table 5.1 Typical Execution Times for Approach 1.

```
Node 0 reports total time on node = 1843
Node 1 reports total time on node = 2801
Node 2 reports total time on node = 3882
Node 3 reports total time on node = 5614
Node 4 reports total time on node = 5627
Node 5 reports total time on node = 5640
Node 6 reports total time on node = 5655
Node 7 reports total time on node = 5837
```

only adds overhead for the processors. One may be able to get a more balanced system by using feedback, but overall goal for this thesis is a faster simulation time.

Since the architecture of the 8-bit carry-lookahead adder consists of two 4-bit carry-lookahead adders, these are better kept as separate entities. The small time difference between node 3 to node 7 confirms that such a decision is reasonable. Therefore, all mappings that are discussed below for nodes 0 through 3 are also applied to nodes 4 through 7.

It is apparent that node 0 could handle more computation, the 3-input AND gate on node 1 is moved to node 0. To avoid feedback, though, every gate which generates a signal that the 3-input AND gate uses for input must also be moved. That means that the first OR gate of the second full adder is transferred from node 1 to node 0. That requires that gate's inputs, X(1) and Y(1), to execute on node 0. Rather than having the behavioral instances for X(1) and Y(1) execute on two processors, it is better to move the gates to which X(1) and Y(1) are attached to node 0, and pass instead signal changes from these latter gates.

Table 5.2 Typical Execution Times for Approach 2.

```
Node 0 reports total time on node = 2089
Node 1 reports total time on node = 3388
Node 2 reports total time on node = 4017
Node 3 reports total time on node = 5330
Node 4 reports total time on node = 5345
Node 5 reports total time on node = 5433
Node 6 reports total time on node = 5451
Node 7 reports total time on node = 5523
```

Similar changes are made for the entire 4-bit carry-lookahead circuit. The result is

the mapping depicted in Figure 5.2. Table 5.2 provides example execution times for the mapping of Figure 5.2.

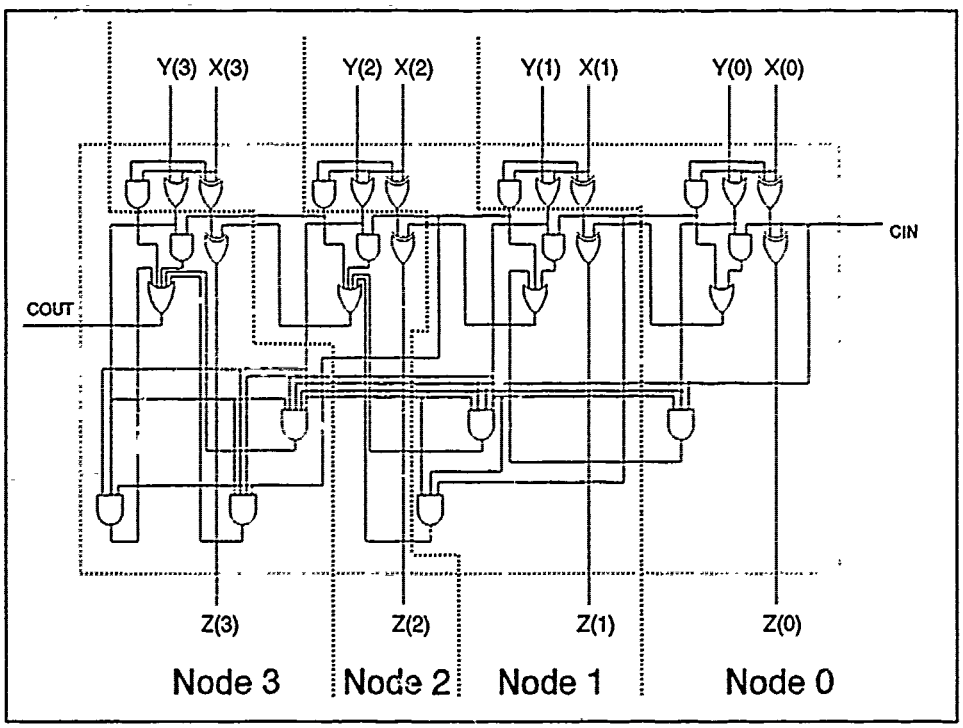

Figure 5.2 Approach 2 for Carry-Lookahead Partitioning.

Table 5.3 Typical Execution Times for Approach 3.

```
Node 0 reports total time on node = 2375
Node 1 reports total time on node = 3321
Node 2 reports total time on node = 3541
Node 3 reports total time on node = 4532
Node 4 reports total time on node = 4555
Node 5 reports total time on node = 4725
Node 6 reports total time on node = 4743
Node 7 reports total time on node = 4788
```

This certainly seems better since node 0 is busier and the total time for the simulation run is faster. To continue this approach of putting more gates (i.e. more computation load) on the upstream processors, a few more gates are moved. The next mapping, called approach 3, is given in Figure 5.3 with example results shown in Table 5.3.

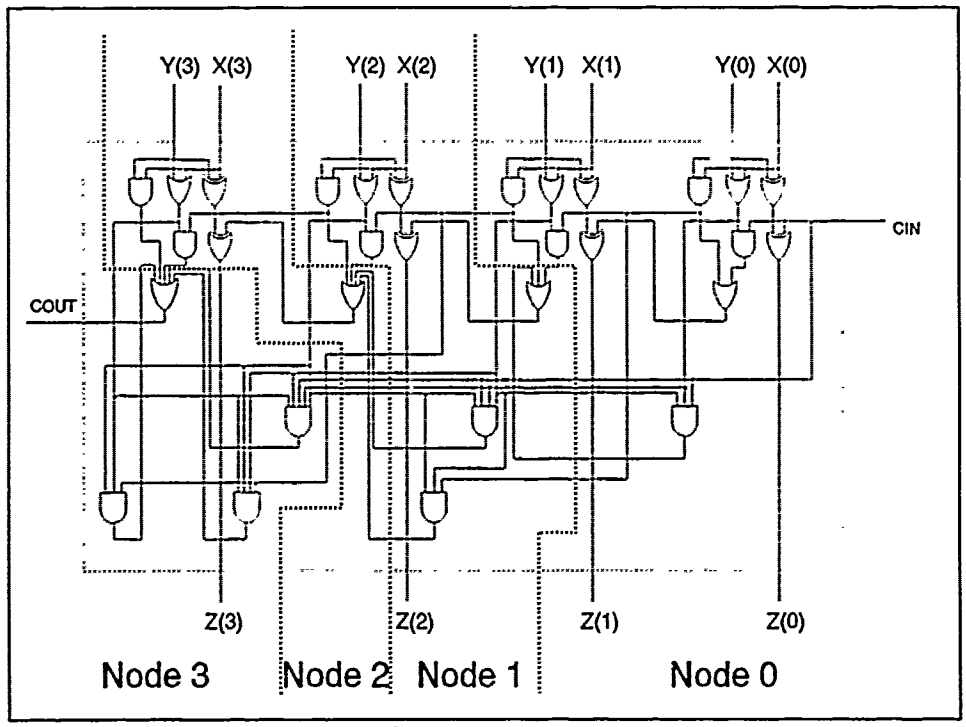

Figure 5.3 Approach 3 for Carry-Lookahead Partitioning.

Although the circuit is more balanced, the execution times between nodes 2 and 3 still seems large. To avoid drastic changes and prevent losing the gains made thus far, only the 3-input AND gate is moved from node 3 to node 2. Since the inputs for the 3-input AND gate are on nodes 0, 1, and 2, no other gates need to be moved. The result is the circuit shown in Figure 5.4 with representative execution times shown in Table 5.4.

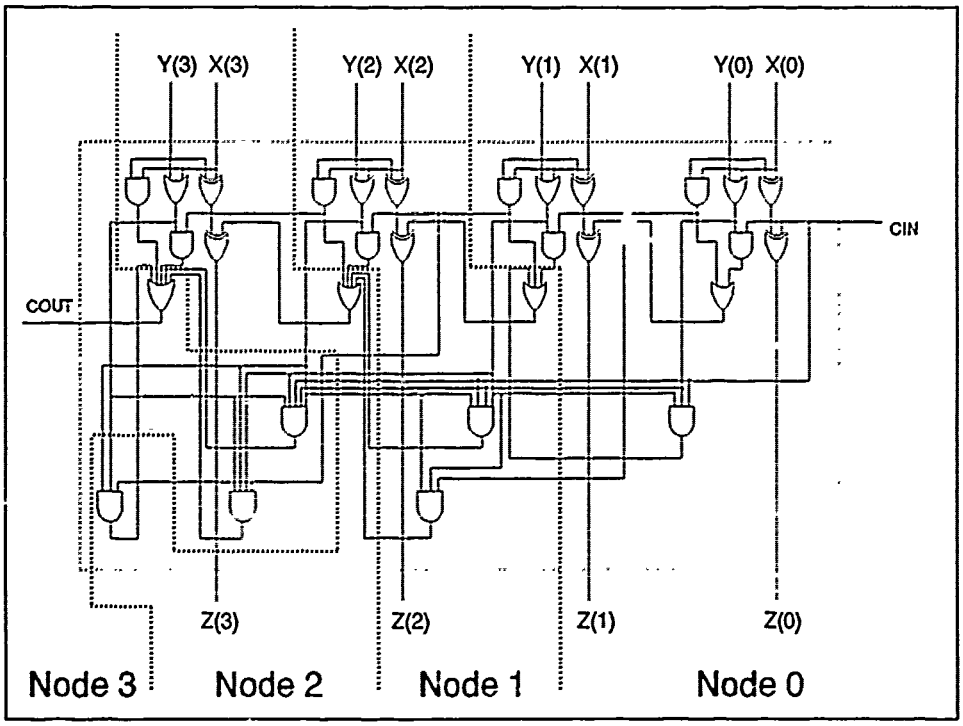

Figure 5.4 Approach 4 for Carry-Lookahead Partitioning.

Table 5.4 Typical Execution Times for Approach 4.

```
Node 0 reports total time on node = 2356
Node 1 reports total time on node = 3344
Node 2 reports total time on node = 3537
Node 3 reports total time on node = 4610
Node 4 reports total time on node = 4639
Node 5 reports total time on node = 4780
Node 6 reports total time on node = 4800
Node 7 reports total time on node = 4814
```

Thirty samples were taken for each mapping strategy and the results are displayed in Table 5.5. Of the four different mappings for the carry-lookahead adder, the fastest execution time is produced by approach three, although approach four is close (within one

standard deviation).

Table 5.5 Statistics for the Four Approaches.

Approach	Mean	Std. Dev.	Max	Min
1	5810	285.248	7091	5597
2	5561	109.617	5932	5433
3	4791	89.214	5032	4652
4	4819	58.120	4900	4706

The explanation for this phenomena is due to basic VHDL algorithm: signals are updated, then any behavioral instances for which the signals are inputs are executed. By recalling that **signals** are the main dependency in the simulation, it is evident that active signals on a node should be minimized if possible. "Active signals" on a node are those which are either an input to a behavioral instance located on the node, or a system output. These are the signals that are entered on the Active Records list. "Total signals" on a node also include signals that are created on the node, but are then sent to another node as input to a behavioral instance there. Table 5.6 shows a comparison of these values for the four approaches discussed above.

Analyzing Table 5.6, one can see that active signals on any given node are minimized with approaches 3 and 4. One reason for approach 3's better performance is that node 2 is required to track 2 fewer signals. It is also evident that the number of gates decrease by one-third as the pipeline is traversed with approach 3 (from 12 to 8 to 6 to 4). Perhaps it is this phenomena which produces better throughput.

While the total number of active signals in the first two approaches is less than the

Table 5.6 Comparison of the Four Approaches.

Approach	Node	Gates	Total Signals	Active Signals
1	0	6	9	7
1	1	7	13	12
1	2	8	16	15
1	3	9	17	17
2	0	10	15	11
2	1	8	17	13
2	2	6	13	11
2	3	6	16	16
3	0	12	17	14
3	1	8	16	15
3	2	6	12	11
3	3	4	12	12
4	0	12	17	14
4	1	8	16	15
4	2	7	12	13
4	3	3	12	12

number in approaches 3 and 4, they are less balanced -- and balance is the key. Further, since changes are propagated through the system, it is better to put a slightly heavier load at the front of the system than at the end. Trying to reduce the number of gates by one-third as one maps a circuit to the hypercube is a simpler paradigm than calculating and minimizing the number of active signals on a node. But, the latter approach is more applicable to a wider range of simulations.

5.5 Logical Processes

Although each behavioral instance has its own independent behavior, they can also be grouped into a larger entity called a "logical process." A logical process (LP), as used in

the parallel VHDL simulator, is simply a group of gates. For example, when the ripple-carry adder is mapped to eight nodes, each full adder is considered a logical process. If, however, one maps the circuit to four nodes, a logical process then consists of two full adders. A logical process is informally defined, therefore, as all of the behavioral instances that reside on any given physical node at a given time.

Logical processes are a concern because, based on the necessary communication between the LPs, a node can tell from which nodes it may receive messages. Without this knowledge, a node finishes processing when its Active Records are exhausted and quits. Then if it receives a message from an upstream node that is still active, events that are supposed to occur, do not. Thus, the simulation is in error.

Therefore, the lpN.arcs file (where "N" is the number of active nodes) is used to depict which LPs are dependent upon which. (The lp.arcs file is more fully discussed in Section 3.8.3.) Knowledge of the interdependence of logical processes is required feature execution under the Chandy-Misra paradigm.

5.6 Output From the Parallel Simulator

In Intermetrics' sequential system the circuit is simulated and the signal changes are written to an internal output file. In order to view the output, one must use their report generator, which analyzes the internal file and prints a report listing changes to the signal values for which the user asked. The report shows bit vectors and scalars on the x-axis in a neat, columnar fashion with increasing time on the y-axis. A portion of one such report for the ripple-carry adder is shown in Figure 5.5.

```
OCT-05-1991 20:34:47          VHDL Report Generator          PAGE 7
  TIME  |--------------------------SIGNAL NAMES--------------------------|

  (NS)      CIN    X(7 DOWNTO 0)    Y(7 DOWNTO 0)    COUT    Z(7 DOWNTO 0)

   600      '0'     "01010101"       "01001100"
   603                                                       "01011111"
   606                                              '0'
   609                                                       "10010001"
   615                                                       "10000001"
   621                                                       "10100001"
   630      '1'     "10101010"       "10110011"
   633                                                       "10100000"
   636                                              '1'
   639                                                       "01101110"
   645                                                       "01111110"
   651                                                       "01011110"
```

Figure 5.5 Sample of Intermetrics' Report.

Note that the report excerpt is for simulation times 600 ns to 651 ns. It shows the results from a VHDL simulation of a ripple-carry adder (Figure 3.19). No entry for a signal at a particular time means that there was no change for that signal. Figure 5.5 is a representation of CIN + X + Y = COUT + Z. Notice how the carry from each full adder "ripples" to the next adder every 3 ns. One can also see that it takes 21 ns from when the inputs are applied to the ripple-carry adder, to when the correct answer is available. This is the "circuit delay" for the full adder and is dependent on the delay times of the gates of which it is composed.

The "X(7 DOWNTO 0)" column shows the bit vector "X" where the first bit is called "X(7)", the next bit is called "X(6)", and so on. "CIN" is the same magnitude as X(0) and Y(0), that is 2^0. "COUT" has the magnitude of 2^8. The equivalent decimal value given by X(n) is its shown binary value ('1' or '0') multiplied by 2^n. Thus, "01010101" has a decimal equivalent of:

$$
\begin{array}{lll}
 & 0 * 2^7 & (= \ \ 0) \\
+ & 1 * 2^6 & (= \ 64) \\
+ & 0 * 2^5 & (= \ \ 0) \\
+ & 1 * 2^4 & (= \ 16) \\
+ & 0 * 2^3 & (= \ \ 0) \\
+ & 1 * 2^2 & (= \ \ 4) \\
+ & 0 * 2^1 & (= \ \ 0) \\
+ & 1 * 2^0 & (= \ \ 1) \ = 85
\end{array}
$$

Unfortunately, the parallel simulator does not yet have a report generator as nice as Int..rmetrics'. Currently, whenever a signal change occurs, a corresponding statement is produced to the screen. One can use the Unix redirect command (">&") to send the screen output to a file (strongly suggested). If the user is just interested in how much time was used on each node, the output statements can be turned off by editing the pvsim.c file, changing the "1" in the line "#define OUTPUT 1" to "0", and recompiling using the "make" command.

```
0---> At time  600 ns, Signal 'Y(0)' is changing from 1 to 0.
3---> At time  570 ns, Signal 'X(3)' is changing from 0 to 1.
4---> At time  555 ns, Signal 'Z(4)' is changing from 1 to 0.
5---> At time  540 ns, Signal 'X(5)' is changing from 1 to 0.
6---> At time  540 ns, Signal 'X(6)' is changing from 0 to 1.
1---> At time  576 ns, Signal 'COUT_0' is changing from 0 to 1.
2---> At time  576 ns, Signal 'COUT_1' is changing from 0 to 1.
7---> At time  459 ns, Signal 'Z(7)' is changing from 1 to 0.
0---> At time  600 ns, Signal 'X(0)' is changing from 0 to 1.
3---> At time  576 ns, Signal 'COUT_2' is changing from 1 to 0.
4---> At time  555 ns, Signal 'COUT_2' is changing from 0 to 1.
5---> At time  543 ns, Signal 'COUT_1' is changing from 1 to 0.
6---> At time  543 ns, Signal 'COUT_1' is changing from 1 to 0.
1---> At time  579 ns, Signal 'Z(1)' is changing from 0 to 1.
2---> At time  579 ns, Signal 'Z(2)' is changing from 0 to 1.
7---> At time  480 ns, Signal 'Y(7)' is changing from 1 to 0.
0---> At time  600 ns, Signal 'CIN' is changing from 1 to 0.
3---> At time  579 ns, Signal 'Z(3)' is changing from 0 to 1.
4---> At time  570 ns, Signal 'Y(4)' is changing from 0 to 1.
5---> At time  558 ns, Signal 'COUT_4' is changing from 0 to 1.
6---> At time  546 ns, Signal 'COUT_5' is changing from 1 to 0.
1---> At time  600 ns, Signal 'Y(1)' is changing from 1 to 0.
2---> At time  600 ns, Signal 'Y(2)' is changing from 0 to 1.
7---> At time  480 ns, Signal 'X(7)' is changing from 1 to 0.
0---> At time  603 ns, Signal 'Z(0)' is changing from 0 to 1.
```

Figure 5.6 Sample Output from an Eight Node Run.

As far as the format of the output from the parallel compiler is concerned, there are four major pieces of information. The first is the node which produced the statement; this is read as the very first character on the line. For example, any line that starts with "0--->" is printed by node 0. Second is the time which is shown in Figure 5.6 in nanoseconds. Third, the signal name is given, such as "Y(0)." Finally, the simulator also shows the change on the signal (e.g., "from 1 to 0").

The output from two or more nodes can be very confusing to read, as shown in Figure 5.6, since the nodes are usually executing at different simulation times. One can, however, use this output to evaluate how far along in the simulation a particular node may be. By noting each node's relative position in the pipeline, one can assess how many test vectors must be entered into the circuit to make parallel execution worthwhile.

To order the output file, one can utilize the Unix sort utility, with a "+1" option, redirected to another file. The "+1" option sorts on the event times rather than on the nodes. Figure 5.7 shows the more orderly output after sorting.

Figure 5.5 and 5.7 provide the same information, however, Intermetrics condenses the information into a better format. In any event, one can track the events of the simulation from either format. For example, Figure 5.7 shows CIN going to '1' at 630 ns; so does Intermetrics' report in Figure 5.5. Figure 5.7 shows Z(1), Z(2), Z(3), and Z(6) going to '0' and Z(7) going to '1' at 609 ns; so does Intermetrics report in Figure 5.5. Therefore, the output from the parallel simulator can be compared to the output from Intermetrics' sequential simulator to verify consistent results.

One can also use the output from either Figure 5.6 or Figure 5.7 to extract the parti-

cular events that occurred on a specific node. For example, using the Unix grep function to extract all strings with a "4--->" in it will extract all events that occurred on node 4. One can also extract specific signals or simulation times, such as "Z(2)" or "600 ns", if desired.

```
0---> At time  600 ns, Signal 'CIN' is changing from 1 to 0.
0---> At time  600 ns, Signal 'X(0)' is changing from 0 to 1.
1---> At time  600 ns, Signal 'X(1)' is changing from 1 to 0.
2---> At time  600 ns, Signal 'X(2)' is changing from 0 to 1.
3---> At time  600 ns, Signal 'X(3)' is changing from 1 to 0.
4---> At time  600 ns, Signal 'X(4)' is changing from 0 to 1.
5---> At time  600 ns, Signal 'X(5)' is changing from 1 to 0.
6---> At time  600 ns, Signal 'X(6)' is changing from 0 to 1.
7---> At time  600 ns, Signal 'X(7)' is changing from 1 to 0.
0---> At time  600 ns, Signal 'Y(0)' is changing from 1 to 0.
1---> At time  600 ns, Signal 'Y(1)' is changing from 1 to 0.
2---> At time  600 ns, Signal 'Y(2)' is changing from 0 to 1.
3---> At time  600 ns, Signal 'Y(3)' is changing from 0 to 1.
4---> At time  600 ns, Signal 'Y(4)' is changing from 1 to 0.
5---> At time  600 ns, Signal 'Y(5)' is changing from 1 to 0.
6---> At time  600 ns, Signal 'Y(6)' is changing from 0 to 1.
7---> At time  600 ns, Signal 'Y(7)' is changing from 1 to 0.
1---> At time  603 ns, Signal 'COUT_1' is changing from 0 to 1.
2---> At time  603 ns, Signal 'COUT_1' is changing from 0 to 1.
5---> At time  603 ns, Signal 'COUT_1' is changing from 0 to 1.
7---> At time  603 ns, Signal 'COUT_1' is changing from 0 to 1.
0---> At time  603 ns, Signal 'COUT_2' is changing from 1 to 0.
0---> At time  603 ns, Signal 'Z(0)' is changing from 0 to 1.
7---> At time  606 ns, Signal 'COUT' is changing from 1 to 0.
1---> At time  606 ns, Signal 'COUT_0' is changing from 1 to 0.
2---> At time  606 ns, Signal 'COUT_1' is changing from 1 to 0.
3---> At time  606 ns, Signal 'COUT_2' is changing from 0 to 1.
6---> At time  606 ns, Signal 'COUT_5' is changing from 1 to 0.
7---> At time  606 ns, Signal 'COUT_6' is changing from 0 to 1.
3---> At time  609 ns, Signal 'COUT_2' is changing from 1 to 0.
1---> At time  609 ns, Signal 'Z(1)' is changing from 1 to 0.
2---> At time  609 ns, Signal 'Z(2)' is changing from 1 to 0.
3---> At time  609 ns, Signal 'Z(3)' is changing from 1 to 0.
6---> At time  609 ns, Signal 'Z(6)' is changing from 1 to 0.
7---> At time  609 ns, Signal 'Z(7)' is changing from 0 to 1.
4---> At time  612 ns, Signal 'COUT_3' is changing from 0 to 1.
4---> At time  615 ns, Signal 'COUT_2' is changing from 0 to 1.
4---> At time  615 ns, Signal 'Z(4)' is changing from 1 to 0.
5---> At time  621 ns, Signal 'Z(5)' is changing from 0 to 1.
0---> At time  630 ns, Signal 'CIN' is changing from 0 to 1.
```

Figure 5.7 Sample Output from an Eight Node Run After Sorting.

5.7 Verification

The output must be verified to assure the user that the simulator is providing the correct result for a given set of inputs. Since the test cases are simulated down to the gate level, one only need verify that the gates are acting as expected to be assured of correct behavior throughout the circuit. For example, since the carry-save adder is made up of eight full adders, eight test cases can be run simultaneously on the eight full adders of the carry-save circuit.

This trait enables one to test every possible set of inputs for a full adder with only two sets of vectors. Since a full adder has three inputs and each input can take on two values ('0' or '1'), there are 2^3, or 8 possible input combinations. Setting X to "01010101," Y to "01001100," and CIN to "0" covers input combinations 000, 010, 011, 100, 101, and 110. Toggling every bit of X, Y, and CIN covers the last two combinations, 001 and 111.

This same testing strategy is used for each of the three test cases. For the carry-save adder, testing was not really necessary since it is made up of the same full adder structure as the ripple-carry adder. Nonetheless, the vectors for X, Y, and Z, provide sufficient coverage to test all eight possible input combinations again.

The carry-lookahead adder consists of full adders and behavioral models for three-, four-, and five-input AND and OR gates. It is senseless to test the full adders again. Therefore, verifying that the models for the new AND and OR gates is the only pertinent concern for the carry-lookahead adder. Since these gates are behavioral models, the inputs are associative; that is, if one input has an effect on the gate, then any of those inputs is sure to have the same effect. Therefore, all that is needed for verification on these gates is to

make sure that they can output '1' and a '0' at the appropriate time for the proper inputs.

Toggling the test vectors "01010101" for X, "01001100" for Y, and '1' for CIN will toggle all of the OR gates. To test the AND gates, all that is needed is to set every bit of X, Y, and CIN to '1'. After the individual delays, each of the AND gates will have a '1' as its output. For a four-bit carry-lookahead adder, changing CIN, Y(0), X(0), Y(1), or X(1) to '0', or changing both X(2) and Y(2), or X(3) and Y(3), to '0' causes at least one AND gate to go to '0'. Thus, several test runs have been completed to verify correct behavior.

As each of the test runs are completed, the resultant vectors are checked to make sure that they correspond to the expected output. Once a behavioral model is verified for a component, any circuit that is made up of those components can be verified as well. Because structural architectures, which are composed of behavioral models, are used for the three test cases, redundant testing can be avoided.

A final point is that although verifying correct output is essential, the intention of this research is to show the feasibility of parallel simulation. Therefore, for timing purposes, inputs to each test case are simply toggled. Since the adders were already verified by the test vectors discussed above, heavy behavioral instance activity became the objective -- not verification. That activity is best achieved by toggling each bit, rather than just one or two.

5.8 Summary

Applying these test cases to the parallel simulator requires several steps. First, one must account for executing the simulation on the hypercube, which involves many concerns. Thes include how to map behavioral instances on the nodes, communication overhead during

the simulation, and implementing the Chandy-Misra paradigm. An efficient implementation of the Chandy-Misra algorithm requires using NULL messages to full advantage without wasting too much time processing irrelevant NULL messages. Finally, output must be produced for the user. This output enables the user to verify that the results from the parallel simulator match the results from Intermetrics' simulator.

As stated throughout this chapter, performance of the parallel simulator depends on several different factors. These include application, parallel architecture, mapping, communication, computational workload, simulation paradigm, and others. Chapter 6 shows how some of these factors are manipulated and reports the results.

VI. Experimentation and Analysis

6.1 Introduction

Running a parallel VHDL simulation offers several choices for the user; all accompanied by both advantages and disadvantages. These choices include the number of nodes to use, how to partition the circuit, communication issues, discrete-event simulation paradigms, and considerations about both grain size and load balancing. Most of these issues have been addressed in the previous chapters. In this chapter, differences in choosing between these issues are compared, the tradeoffs involved are examined, and the work that has been done thus far is analyzed. Finally, directions for future research are submitted for review.

6.2 Experimental Options and Constraints

Currently, the most limiting factor for future researchers is the number of VHDL models available for parallel execution. Therefore, working with only the simulations that are presently out there severely limits the avenues left to be explored. One can vary the number of nodes, vary the computation to communication ratio, change the number of input vectors, or insert spin loops to simulate increased complexity. These variations were tried and the findings are presented in the next section.

By running the simulation on a hypercube, one is forced to use message passing in order to communicate a global variable change from one logical process to another. Because this overhead is so costly, these message channels must be used sparingly. That means that the grain size of the simulation (i.e., how much computation there is to do) must

be quite large. Otherwise, there is no point in porting the simulation to a multiprocessor. However, since the point of trying to parallelize these simulations is to encourage others to follow with much larger and much more complex simulations, the hypercube may prove to be an excellent choice for the long term.

6.3 Comparisons

Although the real thrust of this research is to show that parallel VHDL simulations are realizable, it is important for the reader to see the benefits and disadvantages of parallel execution. Thirty test runs were timed for each data point that appears on the graphs in this chapter and the average of the tests is used as the data point. To insure independence from the host processor and operating system, output was turned off unless stated otherwise. The execution time for a multi-node run is regarded as the longest execution time on any node.

Table 6.1 Statistics on Test Cases.

Adder	Nodes	Mean	Std. Dev.	Min	Max
RCA	1	4796	24.800	4762	4881
RCA	2	2427	69.290	2394	2772
RCA	4	2001	9.030	1982	2029
RCA	8	2200	7.214	2184	2214
CLA	1	10250	19.391	10220	10323
CLA	2	6725	14.292	6707	6772
CLA	4	3892	13.283	3864	3933
CLA	8	4338	53.422	4272	4528
CSA	1	4233	14.566	4208	4277
CSA	2	1932	68.958	1905	2288
CSA	4	1303	27.050	1286	1420
CSA	8	1426	17.145	1380	1442

What follows is a summary of results from tests run on the ripple-carry adder (RCA), the carry-lookahead adder (CLA), and the carry-save adder (CSA) test cases.

Table 6.1 shows the statistics that were gathered for 64 test vectors being fed into each circuit and Figure 6.1 shows how the three test cases match up against each other. Obviously, the carry-lookahead adder is the most taxing in terms of work to be done. However, it also receives the most benefit, in terms of speedup, from distributed processing, as displayed by Figure 6.2.

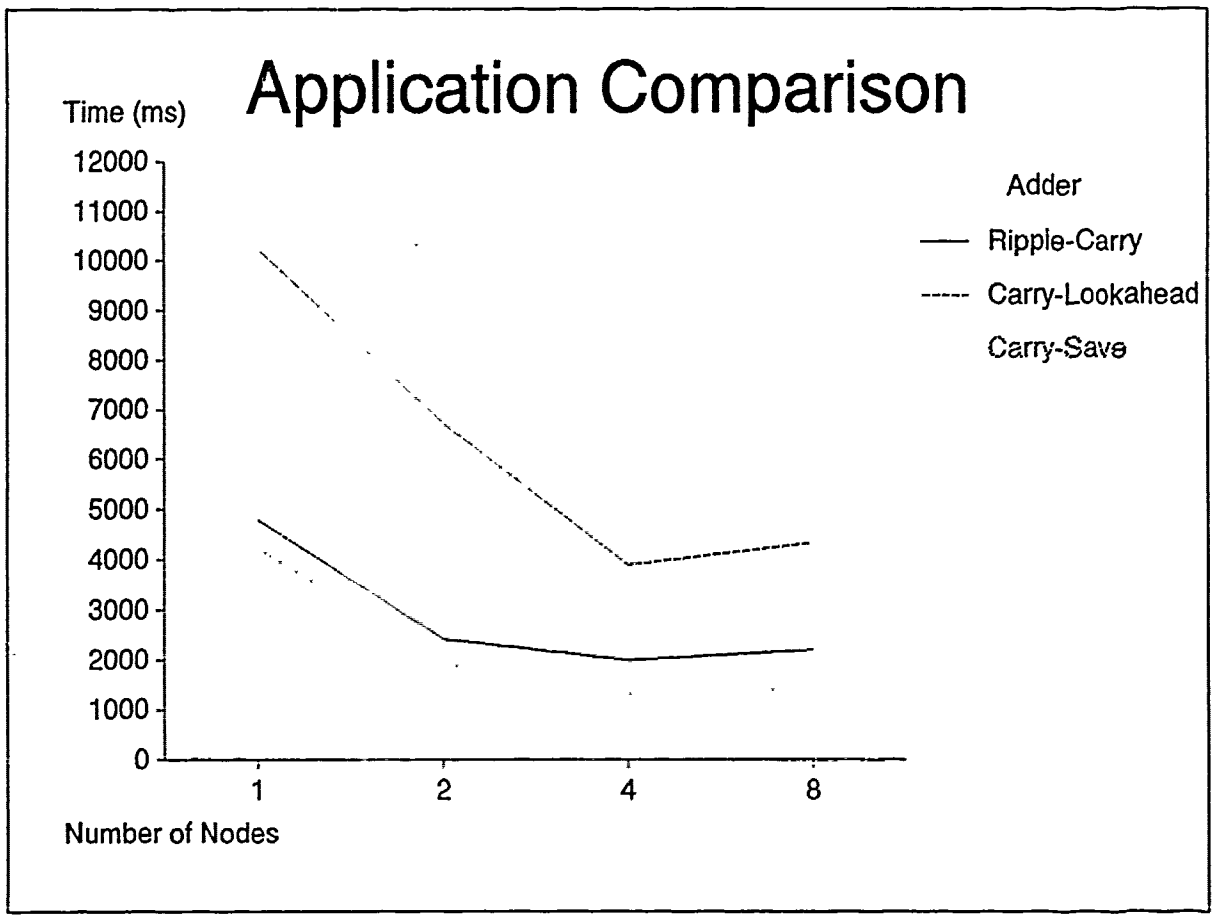

Figure 6.1 Comparison of Applications.

Notice the increase in processing time for the carry-lookahead adder when going from

four nodes to eight nodes in Figure 6.1. This occurs when the computation to communication ratio is decreased, allowing one to deduce that the carry-lookahead adder suffers from too much communication when mapped to eight nodes. Figure 6.2 gives another perspective, showing that speedup is decreased when going from four nodes to eight. A similar effect is appearant in the carry-save adder and ripple-carry adder data. Since there is no communication in the carry-save circuit, one can deduce that the slight increase in execution time is due to the overhead of the host system dealing with eight nodes instead of four (e.g. reading in the lp.arcs and queue.dat files).

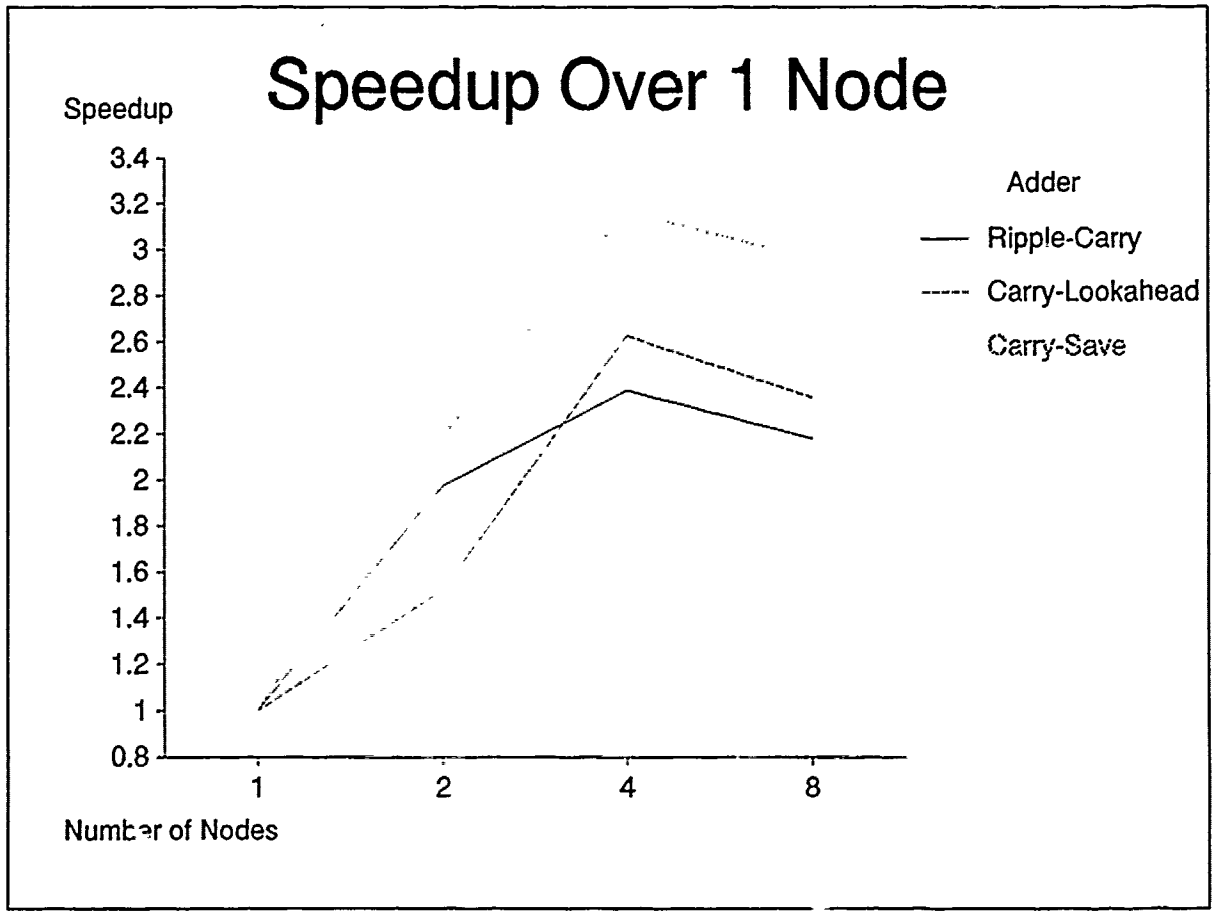

Figure 6.2 Speedup of Applications.

Also of interest is the parallel curves in Figure 6.1 that are plotted for the ripple-carry and carry-save adders. Recall that they are both made up of eight full adders. The ripple-carry adder suffers from the communication costs involved. Accordingly, Figure 6.2 shows speedup for the carry-save adder is greater than that for the ripple-carry adder as the nodes are increased. However, the carry-save adder contains more behavioral instances since three vectors are fed into the full adders instead of just two, as in the ripple-carry adder. In summary, both can be parallelized to a point, but after mapped onto four nodes, any further partitioning is not worth the effort.

In many parallel applications, speedup is achieved by pipelining the data through the processors. Once all of the processors have a portion of the problem to work on, parallel processing begins to pay off. If the processors can be kept busy long enough to offset the overhead of both filling the pipeline and the communication costs, then the user should see a corresponding speedup.

Table 6.2 provides statistics for running the ripple-carry adder with two sets of vectors. Figure 6.3 shows the corresponding plots for average execution time. The top line in the graph depicts 64 test vectors being put through the ripple-carry adder. The bottom line represents 32 test vectors. The graph clearly shows that increasing the number of test vectors for a VHDL simulation makes that simulation a more likely candidate for parallel processing. Also, in the case of pipeline-type simulations, the user must be sure to use enough test vectors to fill the pipeline. Otherwise, parallel simulation is not efficient. If the simulation can be correctly mapped to the architecture, payoff, in the form of speedup, occurs.

Table 6.2 Statistics on Test Vector Test Cases.

Vectors	Nodes	Mean	Std. Dev.	Min	Max
64	1	4796	24.800	4762	4881
64	2	2427	69.290	2394	2772
64	4	2001	9.030	1982	2029
64	8	2200	7.214	2184	2214
32	1	1695	37.864	1515	1747
32	2	1076	14.019	1041	1093
32	4	1182	13.594	1129	1211
32	8	1662	46.555	1520	1834

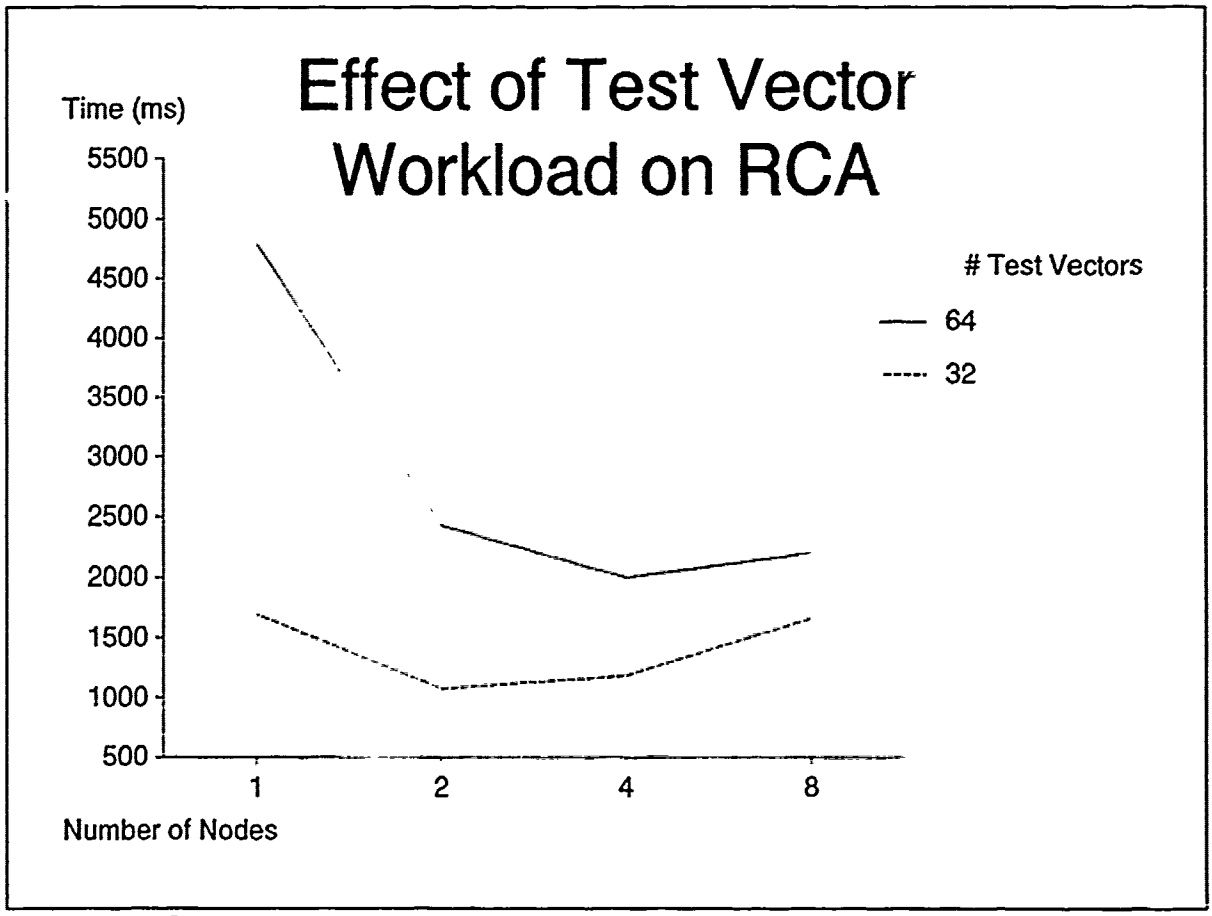

Figure 6.3 Comparison of Effect of Test Vectors.

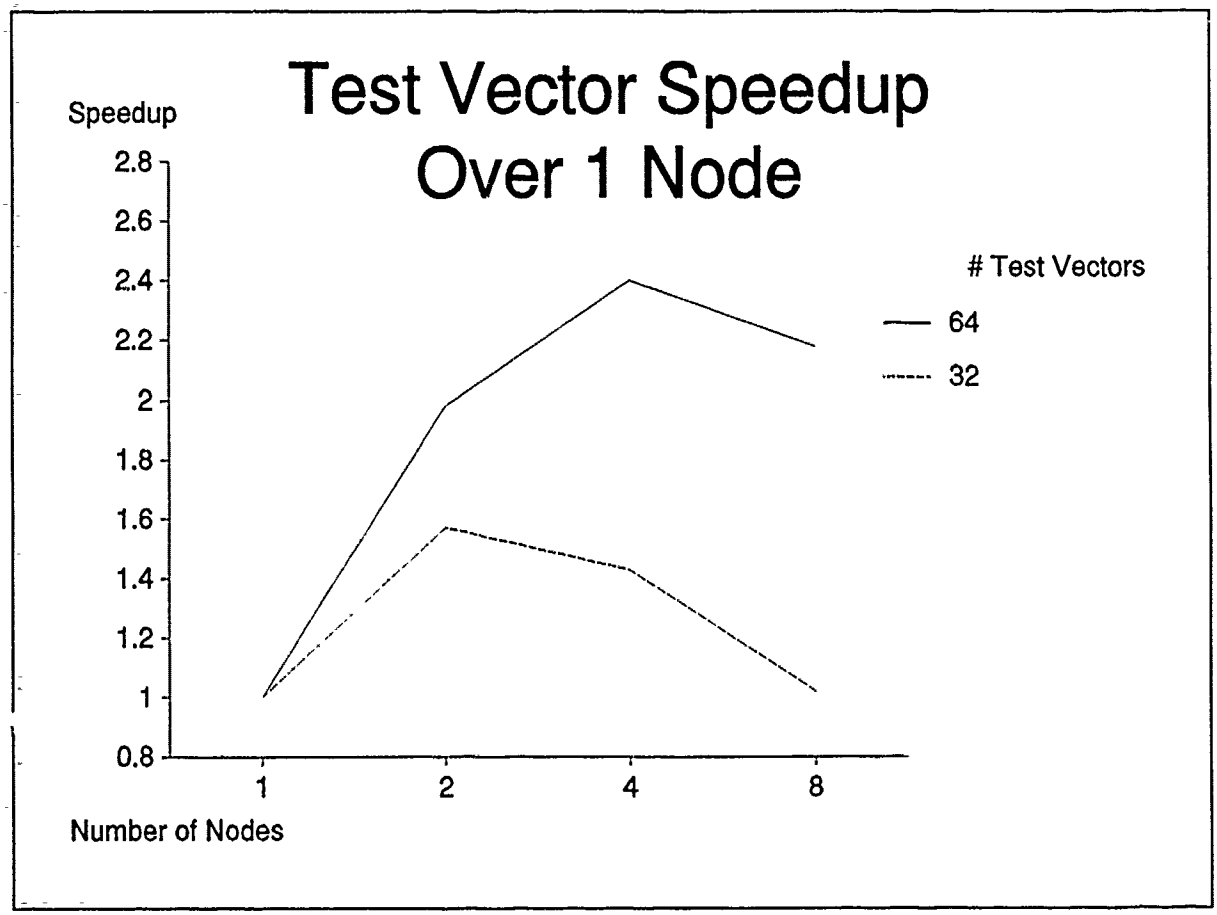

Figure 6.4 Test Vector Effect on Speedup.

Figure 6.4 clearly shows that increasing the number of test vectors in a simulation also increases the benefits of parallelization. This is important since it is logical to expect to simulate larger circuits in the future. Testing larger circuits means putting an increased number of test vectors through the circuit. Figure 6.4 shows that parallel processing is more than capable of handling such a workload.

The test vector experiment (Figure 6.4) increased the computational workload on each node slightly. However, the gains from parallel processing become even more dramatic as the workload is increased by orders of magnitude. This was shown on the hypercube at AFIT by Sartor in 1990 (Sart).

Sartor's research at AFIT showed that as the computational workload of a logical process increases, greater speedup occurs when such applications were mapped to the hypercube. Test runs with the carry-lookahead adder confirm her results, as shown in Figures 6.5. The full statistics are contained in Appendix C.

The spin loop is activated whenever a behavioral instance is executed. That way, although the same number of tasks are being assigned as before, these tasks are artificially being made more intensive. Other experiments can be run by turning the "BUSY" switch on in pvsim.c, finding the two occurrences of the busy loop by locating the word "busy" in pvsim.c, and setting the loop value to whatever is desired.

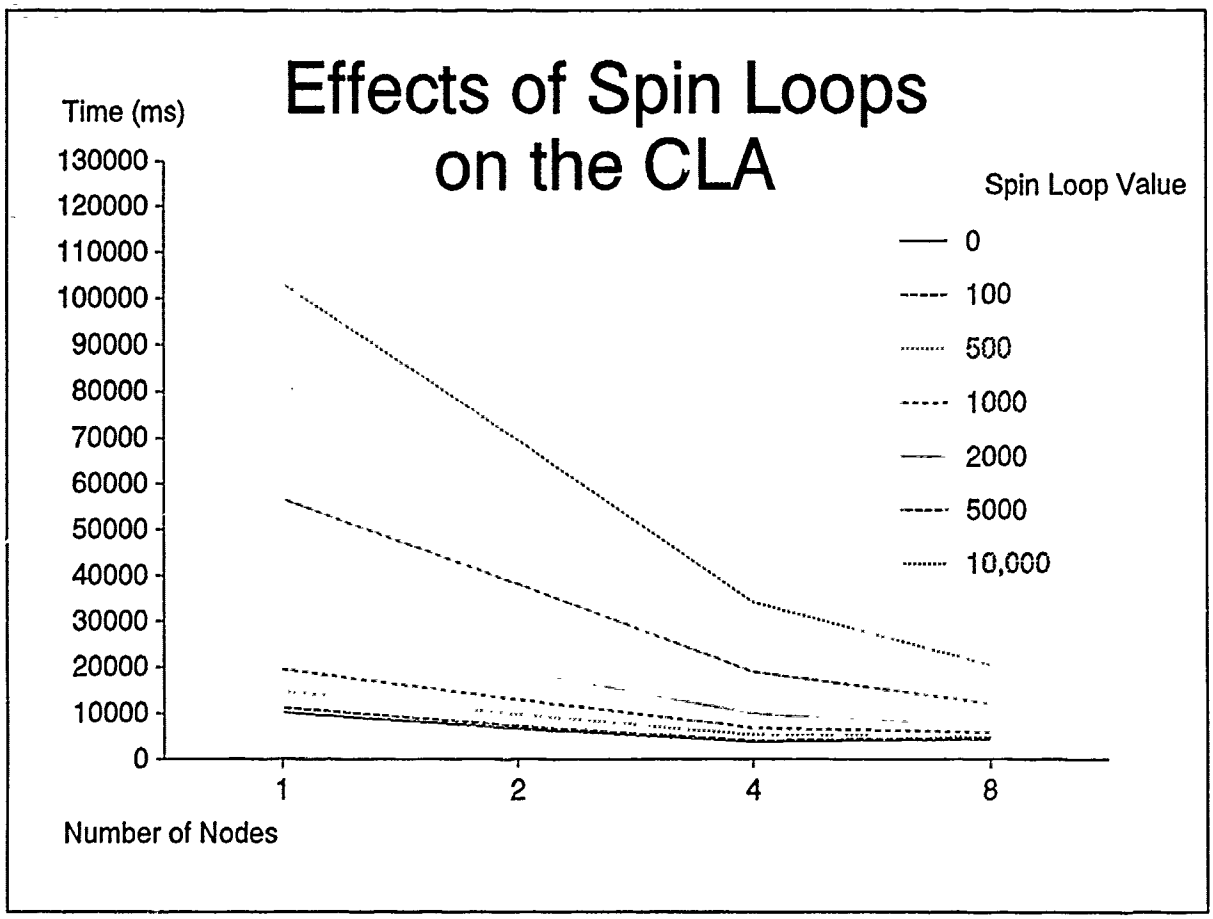

Figure 6.3 Effect of Spin Loops on CLA.

Figure 6.6 shows a different perspective of the spin loop experiment. It shows the speedup that one finds by increasing the workload per behavioral instance. The speedup occurs when the behavioral instances are distributed to several processors. The line corresponding to 10,000 loops indicates that even more processors are needed for such a computational task.

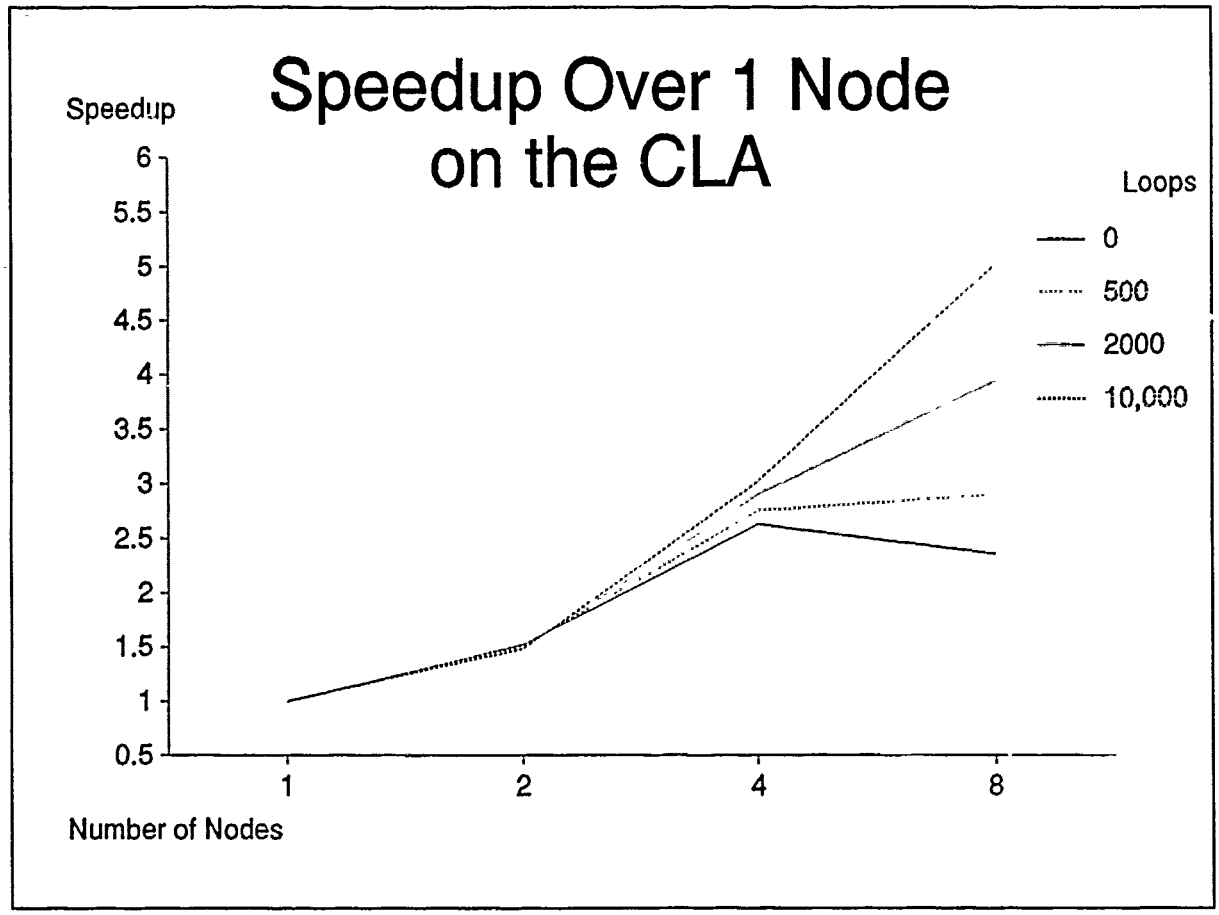

Figure 6.6 Speedup in the CLA From the Spin Loop Effect.

When running an actual simulation, the user is always concerned with the output so that it can be analyzed and verified. Experiments showed that the overhead involved in gathering output is quite significant. Figure 6.7 illustrates the cost of redirecting output

statements to a file. Tests for the ripple-carry adder and the carry-lookahead adder have the same result.

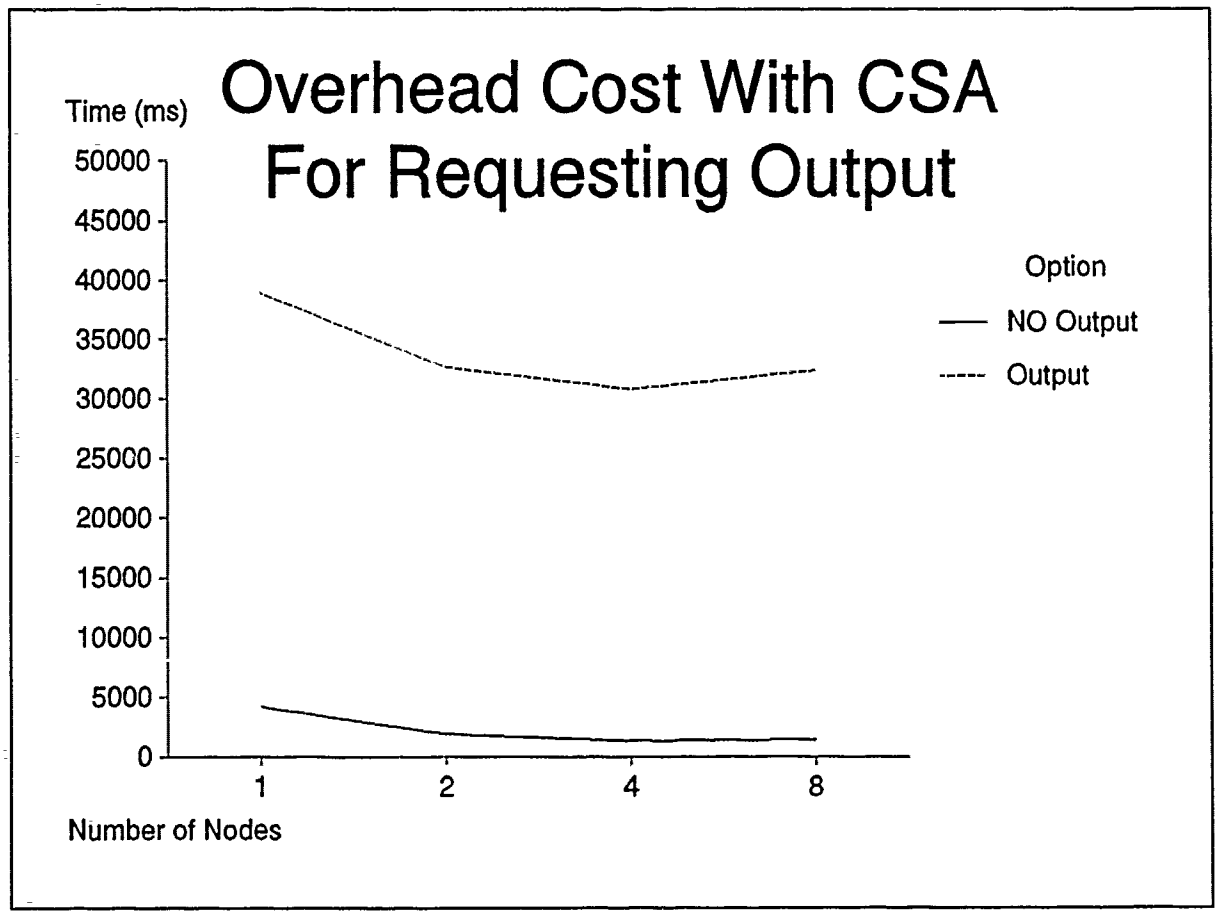

Figure 6.7 Overhead in Processing Output Statements.

Obviously, there is a great difference in the simulation times with output as compared to the times without output. The purpose of Figure 6.7 is to remind the reader to find out how timing was performed before comparing simulation paradigms, architectures, or applications. Since the hypercube requires so much more time to communicate with the file server, times with output will be more varied. Therefore, all timing runs throughout this thesis are done without output. That allows one to infer that time comparisons are

dependent only on the changed variable, and not to a sometimes busy operating system.

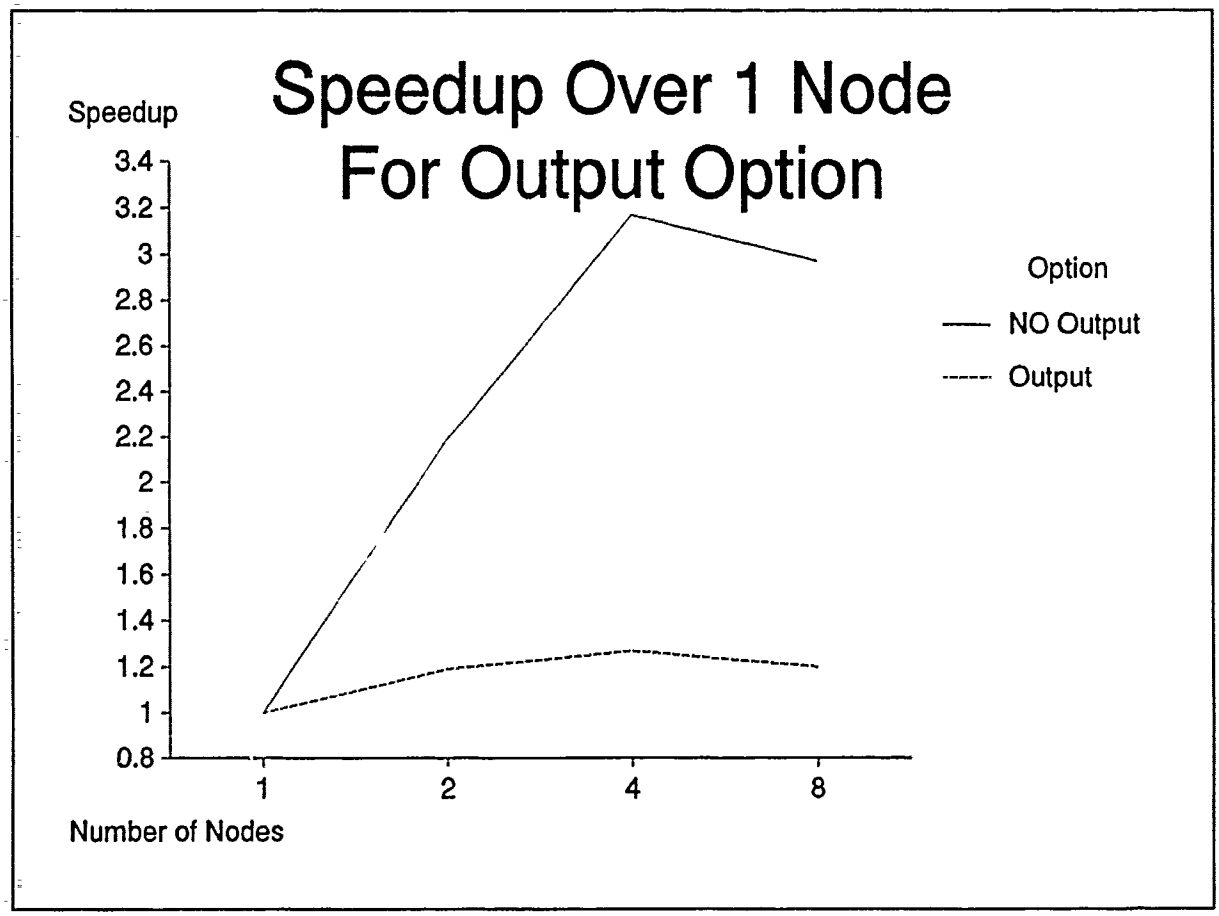

Figure 6.8 Speedup for Output Options.

There is also an effect on speedup if output is turned on. Figure 6.8 indicates that if the nodes must wait on the host operating system for servicing output statements, almost all speedup is lost. This indicates that there are two cases in which parallel simulation feasible (at least for the AFIT iPSC/2). First, parallel simualation is feasible if the simulation is so large and takes so long to run that host service time is negligible. Second, parallel simulation is feasible if the operating system is able to process output request as fast as the nodes can submit them. As 6.8 clearly shows, something must be done so that

advances in speedup do not suffer because of architecture and operating system limitations.

6.4 Analysis of Results

As shown by the above graphs and by the discussions in Chapters 4 and 5, there are many ingredients to successfully porting an application from the sequential world to the parallel. The lessons from the three adder test cases are summarized below. It is assumed that the reader is intending to run parallel VHDL code on a hypercube-type architecture.

Minimize and balance the number of active signals in a logical process. When analyzing a circuit, there is a large temptation to allocate gates to a logical process rather than signals. One must remember that VHDL is signal-driven, not behavior-driven. This guideline may not hold for very complex behaviors, but it should still prove to be a good rule-of-thumb.

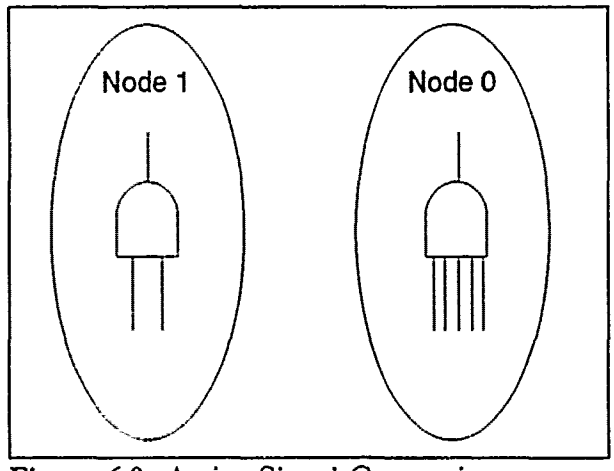

Figure 6.9 Active Signal Comparison.

Note in Figure 6.9 that both nodes have the same number of gates. However, node 0 has five signals which can cause the gate to be executed; node 1 only has two. In the

Intermetrics-based model, if all of the input signals change at once, the gate on node 0 is executed five times. The gate on node 1 is only executed twice. Therefore, in the current model, it is the number of active signals which is the key to distributing workload -- not the number of gates.

Partition a pipeline-type circuit by reducing each nodes work by one-third. This proved to be efficient for the carry-lookahead adder where the workload on four nodes was 12, 8, 6, and 4. Although this worked for this test case, many more experiments should be run to confirm this theory.

Carefully modify the Intermetrics' generated C source code. Until a validated compiler is written, this step in parallel simulation is crucial. It is as difficult to find and correct an error in this stage as it is tedious to perform.

Ensure a high computation to communication ratio. That is, ensure that there are several components on which each node can work or try to reduce communication between nodes. One can try to increase the ratio by better partitioning of the circuit or by consolidating logical processes onto fewer nodes.

6.5 Recommendations for Future Research

There are many interesting areas of research which would further knowledge in this field of parallel VHDL simulation. Since this domain is just beginning to be explored, many of the basics still need to be accomplished. Some ideas are offered below:

1) Write a compiler to translate from Intermetrics' generated C source code to C code which is compatible with the parallel simulator. The compilation process simple, as the

procedure in section 4.2.2 will attest. Until a compiler is written, larger models are not attractive since they will be so much more difficult to "compile by hand."

2) Create and run larger VHDL models. Even greater gains in speedup will be realized once larger, more complex circuits are simulated. Using the carry-save adder, it should be relatively straight-forward to create an eight-bit multiplier.

3) Expand the VHDL subset to include arrays. Often in the world of VHDL there are processes which "sleep" until a certain simulation time arrives. These "sleep times" are recorded as array structures. This research did not successfully implement the necessary array structure and constrained the test cases to those without arrays. This problem simply requires time and a solid understanding of the C programming language (of which this author humbly remains in ignorance).

4) It would also be useful to find some way to automatically partition the VHDL circuit to the nodes. Perhaps one could use the original VHDL configuration file and map the dependencies of components to dependencies of logical processes. Until this is done, one is forced to use trial and error which will not work for an end user.

5) Expand the simulation clock to go past 2.1 seconds. It is currently limited by Intermetrics' model to one 32-bit integer with it value representing a femptosecond. One could follow Intermetrics' lead and concatenate two 32-bit integers to allow for simulation beyond 2.1 seconds, but there may be a more elegant solution.

6) Parallelize a circuit with feedback. This circuit poses many challenges for the user in the areas of mapping, communication, and the optimal number of nodes on which to run. Since no circuits with feedback were considered as part of this research and feedback is a

necessary part of many circuits, research in this area is needed.

6.6 Conclusions

The parallel simulator that has been designed and implemented performs the function for which it was meant. It has successfully simulated three VHDL circuits and demonstrated speedup for each circuit. Although there are constraints in mapping any application to a parallel processor, this simulator and the three test circuits have met those constraints and still performed admirably. VHDL simulations certainly have a future in distributed processing.

Despite great advances in every generation of computers, there has never been a computer that has been called "fast enough." Instead, it is human nature to find bigger and more complex problems which challenge the fastest of the fastest computers. Today, the fastest computer is a parallel processor. But it is only "fast" if the user can map an application to it which is suited for that particular architecture. VHDL applications offer such a mapping for a hypercube-type architecture.

Appendix A. *PVSIM.C Source Code*

This appendix contains a listing of the parallel simulator code which is resident on the iPSC/2 hypercube. The code is commented to help any maintenance programmer understand the design.

A.1 PVSIM.C

```
/*****************************************************************************
 * File Name: pvsim.c  (Parallel Vhdl SIMulator)                             *
 * Author: Capt Ron Comeau                                                   *
 * Advisor: Maj Kim Kanzaki                                                  *
 * Date: 1 Nov 91                                                            *
 * Algorithm:  This file, when linked with generated C source files from     *
 *   the Intermetrics sequential VHDL model generator, will run the VHDL     *
 *   simulation in parallel on the iPSC/2 hypercube.                         *
 *****************************************************************************/

#include "simutl.h"

/* Definitions */
#define EVENT_TYPE 10                   /* VHDL signal event        */
#define MIN_DELAY 3000000               /* min propagation delay    */
#define MAX_TIME 2000000000             /* max simulation time      */
#define REALLOC_BLOCK 1024
#define MAPPING 0                       /* mapping switch (1=on)    */
#define OUTPUT 1                        /* output switch (1=on)     */
#define BUSY 0                          /* spin loop switch (1=on)  */

/* Externals */
extern int my_node;                     /* node id                  */
extern int my_pid;                      /* node process id          */
extern int total_nodes;                 /* total active nodes       */

/* Parallel Processing Variables */
BOOL waiting_on_others = 0;             /* node shutdown flag       */
char siglst[100];                       /* array of pointers to
                                           signal records           */

/* VHDL simulation variables */
int  *cv;                               /* mem location of current
                                           signal values            */
INITK *initst = NULL;                   /* initialize set           */
char *lv;                               /* last value memory        */
BOOL cnstrt = 0;                        /* constraint flag          */
INT32 lvcnt = 0;                        /* connection count         */
TIME  *now;                             /* delta time value         */
int numbeh = -1;                        /* number of behs           */
int numnet = -1;                        /* number of net behs       */
int numsig = -1;                        /* number of net behs       */
TMPK  *tmpbeh;                          /* holder for behs          */
TMPK  *thisptr;                         /* holder for behs          */
```

```
TMPNK *tmpnet;                          /* holder for net behs     */
int coff  = 0;                          /* offset for cv           */
int poff  = 0;                          /* offset for pv           */
int sigid = 0;                          /* next signal id          */
STIME *currnt;                          /* holder for current time */

typedef struct SIG_RECS {               /* Active Record structure */
       int time;                        /* signal change time      */
       SRREC * sr_ptr;                  /* signal record           */
       int   value;                     /* new value               */
       struct SIG_RECS * next_sig_rec;  /* ptr to next active rec  */
} SIG_REC;

typedef struct  EVENTS {                /* Event Record structure  */
        int             from_lp;        /* lpid of lp sending event */
        int             to_lp;          /* lpid of destination lp  */
        int             time;           /* current time of sending lp */
        int             sr_num;         /* sr number               */
      int         sr_time;              /* timestamp of signal change */
      int         value;                /* new value for signal    */
} EVENT;

SIG_REC * sig_rec_head;                 /* ptr to Active Rec list  */
EVENT   * this_event;                   /* ptr to Event record     */
SRREC *srrec_ptr[300];                  /* array of pointers for
                                           every signal in simulation */
int bi_node[100];                       /* array of node location for
                                           every behavioral instance */
CONNT   *check_ptr;                     /* ptr to BI connections   */
int low_time;                           /* lowest time on Active Recs*/
int safe_time[8];                       /* safe time from other nodes*/
BOOL waiting[8];                        /* nodes waiting for messages*/
char outstate[80];                      /* output statement to confirm
                                           what was received by node */


/*************************************************************************
 * Function_Name: init_cv                                                *
 * Purpose:  Performs initialization functions for the parallel simulator. *
 *   Reads in queue.dat and lp.arcs files.  Initializes "waiting" and    *
 *   "safe_time" for each node according to the information in the lp.arcs *
 *   file.  Allocates memory space for current signal values and assigns  *
 *   that pointer to cv.                                                 *
 *************************************************************************/

init_cv ()
{
FILE *fp;
int bi_num, node_num, type_num, from_lp, to_lp, j;
char filename[11];

currnt->delt = 0;
this_event = (EVENT *) malloc(sizeof(EVENT));
check_ptr = (CONNT *) get_new(sizeof(CONNT));

sprintf(filename, "queue%d.dat", total_nodes);  /* read queue.dat file */
if ((fp = fopen(filename, "r")) == NULL)
  {
  printf("Cannot open file %s.\n", filename);
  exit(1);
  }
while (fscanf(fp, "%d %d %d", &bi_num, &node_num, &type_num) >= 0)
  {
  if (type_num == -1) bi_node[bi_num] = my_node;
```

```
  else bi_node[bi_num] = node_num;
  }
if (fclose(fp) != 0 ) printf("error closing queue file.\n");

sprintf(filename, "lp%d.arcs", total_nodes);    /* read lp.arcs file */
if ((fp = fopen(filename, "r")) == NULL)
  {
     printf("Cannot open file %s.\n", filename);
     exit(1);
  }
for (j=0; j<total_nodes; j++)
  {
  safe_time[j] = MAX_TIME;
  waiting[j] = FALSE;
  }
while (fscanf(fp, "%d %d", &from_lp, &to_lp) >= 0)
  {
  if (to_lp == my_node) safe_time[from_lp] = MIN_DELAY;
  if (from_lp == my_node) waiting[to_lp] = TRUE;
  }
safe_time[my_node] = MIN_DELAY;
if (fclose(fp) != 0 ) printf("error closing lp.arcs file.\n");

Z4000001 = (Z4000001_int_struct *) malloc(sizeof(Z4000001_int_struct));
Z4000001->Z0000001_1I400 = 0;
now = &(*currnt->time);                    /* init current time */
return get_new((unsigned)REALLOC_BLOCK);  /* return mem space for
                                          current signal values */
};


/*************************************************************************
 * Function_Name: sim_it                                                 *
 * Purpose:  To execute the behavioral instances that are in the         *
 *   behavior list.  After the last has been executed, the changes to the *
 *   signal values are updated.                                          *
 *************************************************************************/

sim_it (thisptr)
TMPK  *thisptr;
{
int i;
BOOL  update_flag = FALSE;

if (thisptr == '\0')
  { thisptr = tmpbeh;                          /* point to top of Behavior
                                          Records list           */
   update_flag = TRUE;
  }
if (thisptr != '\0')
  {
    if ((*thisptr).nextb != '\0')
      {
      sim_it ((*thisptr).nextb);          /* recursively call sim_it
                                          for next behavior       */
      i = ((*(*thisptr).beh).id);
      if (MAPPING) printf("%d->Executing behavior number %d.\n", my_node, i);
      (*thisptr->beh->exec)((*thisptr).beh);    /* execute behavior        */
      if (BUSY) for (i=0; i<10000; i++) NULL;   /* busy loop               */
      }
    else
      {
      i = ((*(*thisptr).beh).id);
      if (MAPPING) printf("%d->Executing behavior number %d.\n", my_node, i);
```

```
      (*thisptr->beh->exec)((*thisptr).beh);     /* execute behavior        */
      if (BUSY) for (i=0; i<10000; i++) NULL;    /* busy loop               */
      }
    if (update_flag)
      {
      update ();
      }
  }
};


/*****************************************************************************
 * Function Name:  update                                                    *
 * Purpose:  To update the signal values for the next event time and         *
 *   schedule the behavioral instances that they are connected to for        *
 *   execution.  Before the simulation time is advanced, this node must      *
 *   be sure that it will not receive any messages for an earlier time       *
 *   from any upstream nodes.                                                *
 *****************************************************************************/

/* Function Name: update    added by RCC */
update ()
{
SIG_REC * markptr;
SIG_REC * lastmptr;
SIG_REC *templ;
TMPK * hold;
CONNT * behav_ptr;
char units_out[4];
int i, j, time_out, k;
BOOL okay, not_answered, not_safe;

tmpbeh = NULL;
if (sig_rec_head != '\0')                        /* there are still updates
                                                 to make           */
  {
  waiting_on_others = FALSE;
  markptr = sig_rec_head;
  low_time = (*markptr).time;
  while (markptr != '\0')                        /* check all Active Records
                                                 to find the low time    */
    {
    if ((*markptr).time < low_time)
      low_time = (*markptr).time;
      markptr = (*markptr).next_sig_rec;
    }
  safe_time[my_node] = low_time;
  get_event();                                   /* check safe times for all
                                                 other nodes until we can
                                                 update signals          */
  if (currnt->time->least != safe_time[my_node])
    {
    if (safe_time[my_node] >= low_time)
      {
      currnt->time->least = low_time;
      }
    this_event->from_lp = my_node;
    this_event->time = currnt->time->least + MIN_DELAY;
    this_event->sr_num = my_node;
    this_event->sr_time = currnt->time->least + MIN_DELAY;
    this_event->value = -1;
    for (j=0; j<total_nodes; j++)
      {
      if (waiting[j])
```

```
        {                               /* send a null message to
                                           any waiting node       */
        this_event->to_lp = j;
        csend(EVENT_TYPE, this_event, sizeof(*this_event), j, my_pid);
        }
      }
    }
  okay = TRUE;
  for (j=0; j<total_nodes; j++)
    {
    if ( (j != my_node) && (safe_time[j] <= low_time) )
                                    /* */
      okay = FALSE;
    }
  markptr = sig_rec_head;
  lastmptr = sig_rec_head;
  while ( (okay) && (markptr != '\0') && (low_time <= currnt->time->least) )
    {
    if ((*markptr).time == low_time)
      {
      if ((*markptr).value != *(cv + (*markptr).sr_ptr->cval))
      {
        time_out = currnt->time->least;
        if (time_out<1000) sprintf(units_out, "fs");
      else
        {
          time_out = time_out/1000;
          if (time_out<1000) sprintf(units_out, "ps");
        else
          {
            time_out = time_out/1000;
            sprintf(units_out, "ns");
            }
        }
      if (OUTPUT)
        {
        if (time_out>999) printf("%d---> At time %d", my_node, time_out);
        else
          {
          if (time_out>99) printf("%d---> At time  %d", my_node, time_out);
          else printf("%d---> At time   %d", my_node, time_out);
          }
          if (currnt->delt == 0) printf(" %s,", units_out);
          else printf(" %s + %d,", units_out, currnt->delt);
        printf(" Signal '%s' is changing from %d to %d.\n",
           (*markptr).sr_ptr->name, *(cv + (*markptr).sr_ptr->cval),
           (*markptr).value);
        }

      *(cv + ((*markptr).sr_ptr->cval)) = (*markptr).value;
        behav_ptr = (*markptr).sr_ptr->conns;
        while (behav_ptr != '\0')
          {
          i = ((*(*behav_ptr).bhv).id);
        if (bi_node[i] == my_node)
          {
            hold = (TMPK *) get_new(sizeof(TMPK));
            hold->nextb = tmpbeh;
            hold->beh = beh_ptr->bhv;
            tmpbeh = hold;
            }
          behav_ptr = behav_ptr->nb;
        }
      }
```

```
      if (sig_rec_head == markptr) sig_rec_head = (*markptr).next_sig_rec;
      markptr = (*markptr).next_sig_rec;
      (*lastmptr).next_sig_rec = markptr;
      }
      else
      {
      lastmptr = markptr;
      markptr = (*markptr).next_sig_rec;
      }
      }
    if (sig_rec_head == '\0')
      {
      waiting_on_others = FALSE;
      for (j=0; j<total_nodes; j++)
        if ( (j != my_node) && (safe_time[j] < MAX_TIME) )
        waiting_on_others = TRUE;
      while (waiting_on_others)
      {
        get_event();
      if (tmpbeh == '\0') update();
      sim_it (NULL);
        waiting_on_others = FALSE;
        for (j=0; j<total_nodes; j++)
          if ( (j != my_node) && (safe_time[j] < MAX_TIME) )
          waiting_on_others = TRUE;
      }
    }
  if (tmpbeh == '\0') update();
  sim_it (NULL);
  }
}


/*****************************************************************************
 * Function_Name: get_event                                                  *
 * Purpose:  To receive messages from upstream nodes until the simulation    *
 *           can proceed.                                                    *
 *****************************************************************************/
get_event()
{
SIG_REC *temp1;
char units_out[4];
int i, j, k;
BOOL not_answered;

  safe_time[my_node] = low_time;
  i=0;
  while (i < total_nodes)
      {
      if ( ( (safe_time[i] <= low_time) && (i != my_node) )
         || ( (waiting_on_others) && (safe_time[i] < MAX_TIME) &&
              (i != my_node) ) )
        {
        not_answered = TRUE;
        while (not_answered)
          {
          crecv(EVENT_TYPE, this_event, sizeof(*this_event) );
          if (this_event->value == -1 )
            {
            j = this_event->from_lp;
            if ((safe_time[j]) < (this_event->time))  /* update safe time */
              {
              safe_time[j] = this_event->time;
              }
```

```
            if ((this_event->time) == -1)      /* that node is done */
              {
              safe_time[j] = MAX_TIME;
              waiting_on_others = FALSE;
              for (j=0; j<total_nodes; j++)
                if ( (j != my_node) && (safe_time[j] < MAX_TIME) )
                waiting_on_others = TRUE;
            }
            not_answered = ( j != i );
            safe_time[my_node] = safe_time[j];
            for (j=0; j<total_nodes; j++)
              if ((safe_time[my_node]) > (safe_time[j]))
                {
                  safe_time[my_node] = safe_time[j];
                }
            if (currnt->time->least < safe_time[my_node])
                {
              if (safe_time[my_node] < low_time)
                {
                  currnt->time->least = safe_time[my_node];
                }
              else
                {
                currnt->time->least = low_time;
                }
              this_event->from_lp = my_node;
              this_event->time = currnt->time->least + MIN_DELAY;
              this_event->sr_num = my_node;
              this_event->sr_time = currnt->time->least + MIN_DELAY;
              this_event->value = -1;
              for (j=0; j<total_nodes; j++)
                {
                if (waiting[j])
                  {
                  this_event->to_lp = j;
              csend(EVENT_TYPE, this_event, sizeof(*this_event), j, my_pid);
                  }
                 }
                }
              }
            else
              {
              templ = (SIG_REC *) malloc(sizeof(SIG_REC));
              templ->time = this_event->sr_time;
              k = this_event->sr_num;
              templ->sr_ptr = srrec_ptr[k];
              templ->value = this_event->value;
              templ->next_sig_rec = sig_rec_head;
              sig_rec_head = templ;
            sprintf(outstate, "%d->Rcvd %s with value %d for time %d\0",
            my_node, templ->sr_ptr->name, templ->value, templ->time);
              if (low_time > (this_event->sr_time))
                {
                low_time = this_event->sr_time;
                }
              }
            if ((this_event->from_lp) == i) not_answered = FALSE;
            }
          }
        i++;
        }
}
```

```
/*****************************************************************************
 * Function_Name: mksig                                                      *
 * Purpose:  To allocate a signal record for every signal, assign            *
 *           initial values to the record's fields, add the pointer to       *
 *           the record in the correct position of the srrec_ptr array,      *
 *           and return the pointer to the calling program.                  *
 *****************************************************************************/
  SRREC *
mksig (sz)
  INT16 sz;
{
  SRREC *sr;

  sr = (SRREC *) get_new(sizeof(SRREC));
  sr->id = sigid;
  sr->size = sz;
  sr->trp = FALSE;
  sr->attp = FALSE;
  sr->disc = FALSE;
  sr->atts = NULL;
  sr->cval = coff;    /*  added by RCC  */
  sr->trans = NULL;
  sr->conns = NULL;
  sr->descr = NULL;
  poff = poff + sz;
  coff = coff + sz;
  numsig = sigid;
  srrec_ptr[sigid] = sr;
  sigid++;
  return sr;
};


/*****************************************************************************
 * Function_Name: strbi                                                      *
 * Purpose:  To add behavioral instances to the behavior list.               *
 *****************************************************************************/
strbi (bi, tzp, sr)
         BHINST *bi;
         BOOL   tzp;
         SRREC  *sr;
{
  TMPK  *hold;
  TMPNK *thold;

  if (bi->prty >= tstb) {
    if (tmpnet == NULL || tmpnet->beh != bi) {
      numnet++;
      bi->id = numnet;
    };
    thold = (TMPNK *) get_new(sizeof(TMPNK));
    thold->nextb = tmpnet;
    thold->sig = sr;
    thold->beh = bi;
    tmpnet = thold;
  } else {
    numbeh++;
    bi->id = numbeh;
    bi->stkrec.lngth = -1;
    bi->stkrec.stack = NULL;
    if (bi_node[numbeh] == my_node)
     {
      hold = (TMPK *) get_new(sizeof(TMPK));
```

```
      hold->nextb = tmpbeh;
      hold->beh = bi;
      tmpbeh = hold;
      )
  );
  return XNONE;
);

/*****************************************************************************
 * Function_Name: setkck                                                     *
 * Purpose:  To connect a behavioral instance output to any behavioral       *
 *           instances for which it is an input.  When an output value       *
 *           changes, the behavioral instances which will be affected are    *
 *           added to the behavior list (hold) for execution. Those BIs      *
 *           are found via the pointers in the "conns" field.                *
 *****************************************************************************/
  CONNT *
setkck (sr, bi, flagstat)
  SRREC  *sr;
  BHINST *bi;
  BOOL flagstat;
{
  CONNT *hold,*tmp;

  hold = (CONNT *) get_new(sizeof(CONNT));
  hold->bhv = bi;
  hold->flag = flagstat;
  if (bi->prty > tstb) {
    hold->nb = sr->conns;
    sr->conns = hold;
  ) else {
    hold->nb = NULL;
    tmp = sr->conns;
    if (tmp == NULL) {
      sr->conns = hold;
    ) else {
      while (tmp->nb != NULL) tmp = tmp->nb;
      tmp->nb = hold;
    );
  );
  return hold;
);

/*****************************************************************************
 * Function_Name: init                                                       *
 * Purpose:  To initialize every BI that a signal is an input for and        *
 *           execute it to make sure its outputs have a correct value for    *
 *           the newly initialized input.                                    *
 *****************************************************************************/
init(fnc, pktl, brbl)
         FNP    fnc;
       LSTNDE *pktl;
         LSTNDE *brbl;
{
         INITK *hold;
         INITK *initptr=initst;

  hold       = (INITK *) malloc(sizeof(INITK));
  hold->func = fnc;
  hold->pkts = pktl;
  if (initptr && initptr->pkts == NULL) {
    while (initptr->nexti && initptr->nexti->pkts == NULL) {
     initptr = initptr->nexti;
```

```
    };
    hold->nexti = initptr->nexti;
    initptr->nexti = hold;
  } else {
    hold->nexti = initst;
    initst = hold;
  };
  (*(&fnc))(pktl);
  return XNONE;
};


/*****************************************************************************
 * Function_Name:  padit                                                     *
 * Purpose:  To increment the offset values of memory pointers.              *
 *****************************************************************************/

padit (sz)
  UINT16 sz;
{
  poff = poff + sz;
  coff = coff + sz;
}


/*****************************************************************************
 * Function_Name: strsr                                                      *
 * Purpose:  Intermetrics uses it for memory alignment of all of the         *
 *           the signal records.  I use it because this is where each          *
 *           signal is assigned a unique I.D.                                *
 *****************************************************************************/
strsr(sr)
SRREC *sr;
{
#ifdef SUN4
{
    /* SAB (4/12/90)
       The Sun 4 requires strict memory alignment for the appropriate data
       types.  Therefore, padding may be necessary for various signal types
       depending on the current value of coff.
    */
    int padding = coff % sr->size;
    int lcv;
    char *old_cv;

    if (sr->size != 1) {
      if (padding != 0) {
        coff += (sr->size - padding);
      }
    }
    /* Ensure the signal current value array (cv) is large enough, if not
       reallocate it.
    */

    if ((coff + sr->size) >= save_coff) {
      /* cv = realloc(cv, (unsigned)REALLOC_BLOCK); /* realloc on Sun 4 buggy */
      old_cv = cv;
      /* Allocate new memory block */
      cv = get_new((unsigned)(save_coff + REALLOC_BLOCK));
      /* Copy contents of old signal value array to new one */
      cv = memcpy(cv, old_cv, coff);
      /* Initialize the unitialized part of new array to 0's */
      for (lcv=coff; lcv<(save_coff+REALLOC_BLOCK); lcv++)
        cv[lcv] = 0;
```

```
      /* Bump save_coff to point to end of the new block */
      save_coff += REALLOC_BLOCK;
      /* Release old signal array memory block */
      free(old_cv);
    }
}
#endif
    sr->cval = coff;
    coff += sr->size;
    if (sr->attp)
    {
      sr->atts = (STQT *) get_new(sizeof(STQT));
      sr->atts->lstevnt = (STIME *) get_new(sizeof(STIME));
      sr->atts->lsttrns = (STIME *) get_new(sizeof(STIME));
      sr->atts->lstevnt->time = (TIME *) get_new(sizeof(TIME));
      sr->atts->lsttrns->time = (TIME *) get_new(sizeof(TIME));
#ifdef SUN4
{
        /* SAB (4/17/90)
           lval and lvcnt must also ensure proper alignment on the
           Sun 4.
        */
        int padding = lvcnt % sr->size;
        if (sr->size != 1) {
          if (padding != 0) {
            lvcnt += (sr->size - padding);
          }
        }
}
#endif
      sr->atts->lval = lvcnt;
      lvcnt = lvcnt + sr->size;

        sr->atts->lstevnt->time->least = -2;
        sr->atts->lstevnt->time->most = 2147483647;
        sr->atts->lstevnt->delt = 0;
        sr->atts->lsttrns->time->least = -2;
        sr->atts->lsttrns->time->most = 2147483647;
        sr->atts->lsttrns->delt = 0;
    }
    siglst[sr->id] = sr;
}

/*****************************************************************************
 * Function_Name:  sqadd                                                     *
 * Purpose:  To add to times together (I followed Intermetrics example --    *
 *           don't ask me why they do this twice.)                             *
 *****************************************************************************/
sqadd (time1, time2, time3)
TIME * time1, time2, time3;
{ (*time1).least = time2.least + time3.least;
  (*time1).least = time2.least + time3.least;
};


/*****************************************************************************
 * Function_Name:  posts8                                                    *
 * Purpose:  To post an event (new signal output) for a future time.         *
 *****************************************************************************/
posts8 (pointr1, number1, addr1, type1, constrt1)
int  number1, type1, constrt1;
SRP pointr1;
TIME * addr1;
```

```
{
SIG_REC *templ;
int send_msg[8];
int i, j;

check_ptr = pointrl->conns;
for (i=0; i<total_nodes; i++) send_msg[i] = 0;
if (check_ptr == '\0')              /* an output signal is being posted */
  {                                 /* add it to the list */
     templ = (SIG_REC *) malloc(sizeof(SIG_REC));
     templ->time = addrl->least + currnt->time->least;
     templ->sr_ptr = pointrl;
     templ->value = numberl;
     templ->next_sig_rec = sig_rec_head;
     sig_rec_head = templ;
     i= total_nodes + 1;
  }
while (check_ptr != '\0')
  {                             /* identify nodes which are affected by
                                      by the signal change  */
  i = check_ptr->bhv->id;
  j = bi_node[i];
  send_msg[j] = 1;
  check_ptr = ((*check_ptr).nb);
  i=0;
  }
while (i < total_nodes)
  {
  if ( send_msg[i] )
    {
    if (my_node == i)         /* post to this node's active records */
      {
      templ = (SIG_REC *) malloc(sizeof(SIG_REC));
      templ->time = addrl->least + currnt->time->least;
      templ->sr_ptr = pointrl;
      templ->value = numberl;
      templ->next_sig_rec = sig_rec_head;
      sig_rec_head = templ;
      check_ptr = pointrl->conns;
      while (check_ptr != '\0')
        {
        j = check_ptr->bhv->id;
        if (MAPPING && (bi_node[j]==my_node))
             printf("%d-> Add behav %d to my list (node %d) for time %d \n",
             my_node, j, my_node, templ->time);
        check_ptr = ((*check_ptr).nb);
        }
      }
    else
      {
      if (currnt->time->least > 0)        /* send a message to the node */
        {
        this_event->from_lp = my_node;
        this_event->to_lp = i;
        this_event->time = currnt->time->least + MIN_DELAY;
        this_event->sr_num = pointrl->id;
        this_event->sr_time = currnt->time->least + addrl->least;
        this_event->value = numberl;
        csend(EVENT_TYPE, this_event, sizeof(*this_event), i, my_pid);
        }
      }
    }
  i++;
  }
```

```
}


/*****************************************************************************
 * Function_Name:  rptast                                                    *
 * Purpose:  To report an assertion error when one is raised.                *
 *****************************************************************************/
rptast () {
   if (OUTPUT)
     printf("Assertion error at %d\n", currnt->time->least);};


/*****************************************************************************
 * Function_Names: close_sigdict, m_real_type, m_int_type, m_signal, pop,    *
 *                 push, read_input, rmtrrec, rpterr, rptstats, sched,       *
 *                 Start_Nonarray_Comp, timer, and tpop                      *
 * Purpose:  To allow me to leave these calls in the Intermetrics-generated  *
 *           C source code.  They simply return to the calling program.      *
 *****************************************************************************/
close_sigdict () {};
m_real_type () {};
m_int_type () {};
void m_signal(signal, sig_id, dummy)
   int  signal[];   int  sig_id;   int  dummy;     {};
pop () {};
push (str1, number1, pointr1)
      char str1[25]; int  number1;  int  pointr1;  {};
read_input () {};
rmtrrec () {};
rpterr () {};
rptstats () {};
sched () {};
Start_Nonarray_Comp () {};
timer () {};
tpop () {};
```

Appendix B. *An Example*

To help the reader understand how the parallel simulator works, an example is provided. This appendix allows the reader to follow the pseudocode in Figure 4.19 using the full adder circuit. Below tells how the circuit is mapped to and executed on the parallel simulator.

B.1 The Full Adder

The circuit diagram for the full adder is shown in Figure B.1 and the lp.arcs file and the queue.dat file are displayed in Tables B.1 and B.2, respectively.

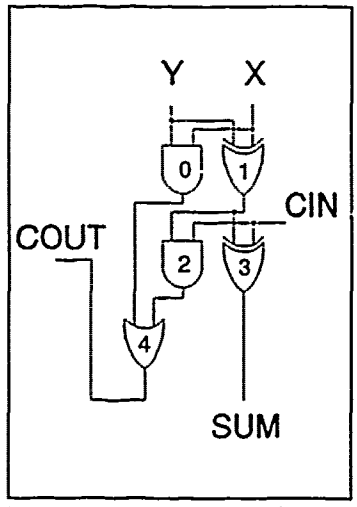

Figure B.1 Full Adder Circuit Diagram.

Table B.1 Lp.arcs File.

From	To
0	1
0	2
1	2

Table B.2 Queue.dat File.

Behavioral Instance	Node
0	0
1	0
2	1
3	1
4	2

After lines 1 through 10 of Figure 4.19 have been executed, Figure B.2 shows the safe_times and the waiting values for each of the three processors. In this application, MAX_TIME is set to 2000 ns and MIN_DELAY is set to 3 ns (the delay of a single gate). At line 11 all of the signals are initialized to a '0' value. The data records are set up as shown in Tables B.3 and B.4. Since the "cval" field in the Signal Record is simply a pointer to a memory location that holds the current value, this example exchanges a current value field, "cur_val", for clarity in the following tables.

Table B.3 Signal Records at Initialization.

id	size	name	cur_val	conns
0	1	Y	'0'	0,1
1	1	X	'0'	0,1
2	1	CIN	'0'	2,3
3	1	COUT_1	'0'	4
4	1	SUM_1	'0'	2,3
5	1	COUT_2	'0'	4
6	1	SUM	'0'	
7	1	COUT	'0'	

Table B.4 Behavioral Instance Records.

id	node	exec	input0	input1	output0
0	0	address(AND)	0	1	3
1	0	address(XOR)	0	1	4
2	1	address(AND)	2	4	5
3	1	address(XOR)	2	4	6
4	2	address(OR)	3	5	7

P0	P1	P2
waiting[1] = T	waiting[0] = F	waiting[0] = F
waiting[2] = T	waiting[2] = T	waiting[1] = F
safe_time[1] = 2000	safe_time[0] = 3	safe_time[0] = 3
safe_time[2] = 2000	safe_time[2] = 2000	safe_time[1] = 3
sim_time = 0	sim_time = 0	sim_time = 0

Figure B.2 Current Processor States.

At line 11, any system input signals which affect any BI on the node are added to the Active Records list, which is shown in Figure B.3. These input signals are supplied by the test bench originally and come as part of the Intermetrics-generated C source. For purposes of this example, let "X" go to '1' at 50 ns. At line 12, the gate behavioral instances executed (if they are assigned to the node). Thus, the Behavior List for each processor at simulation time 0 is as shown in Figure B.4.

P0			P1			P2		
time	value	sigrec	time	value	sigrec	time	value	sigrec
50	'1'	1						

Figure B.3 Current Active Records.

P0	P1	P2
beh	beh	beh
0	2	4
1	3	

Figure B.4 Current Processor States.

After the behavioral instances are executed at lines 14 through 23, the Behavior List is empty on each of the nodes. Active Records have been posted as a result of the executions at

time 0 ns. The result is shown in Figure B.5. The Low_Time on all three nodes is 3 ns, but node 0 is the only one that can process since node 0's safe times are greater than 3 ns. The other wait to receive a message to update their safe times.

Node 0 assigns 3 ns as the simulation time on node 0 and sends this time plus MIN_DELAY to the nodes where the "waiting[]" value is true. Node 1 can now proceed in the same fashion. It updates its simulation time to 3 ns and sends a message to node 2. Node 2 is now free to continue processing also. The new safe time values are shown in Figure B.6.

P0			P1			P2		
time	value	sigrec	time	value	sigrec	time	value	sigrec
50	'1'	1	3	'0'	5	3	'0'	7
3	'0'	3	3	'0'	6			
3	'0'	4						

Figure B.5 Current Active Records.

P0	P1	P2
waiting[1] = T	waiting[0] = F	waiting[0] = F
waiting[2] = T	waiting[2] = T	waiting[1] = F
safe_time[1] = 2000	safe_time[0] = 6	safe_time[0] = 6
safe_time[2] = 2000	safe_time[2] = 2000	safe_time[1] = 6
sim_time = 3	sim_time = 3	sim_time = 3

Figure B.6 Current Processor States.

Proceeding with line 36 on each node, they all have records for 3 ns, but all of the new values match the old values. Therefore, these records are simply discarded to prevent propagating useless calculation. They return to the loop on line 13 and all three enter the loop

and jump to line 24. The Low_Time for node 0 is 50 ns and this is assigned as the Simulation_Time as well. Node 1's safe_time for node 0 forces it to wait for a message at line 26. Node 2 also must wait for a message at line 26.

Node 0 sends a NULL message for 53 ns (simulation time plus MIN_DELAY) to nodes 1 and 2. They both update their safe time for node 0, but they both must continue waiting. Node 0 proceeds up to line 38. Since the new value ('1') is different from the current value, the current value is updated for signal 1. Line 42 uses the "conns" field of signal record 1 to see which BIs are affected. It then cross-references these BIs to their records to see on which node they reside. In both cases it is node 0, so it adds BIs 0 and 1 to node 0's Behavior List. Table B.5 shows the signal values for node 0 at time 50 ns.

Table B.5 Node 0's Signal Records at Time 50 ns.

id	size	name	cur_val	conns
0	1	Y	'0'	0,1
1	1	X	'1'	0,1
2	1	CIN	'0'	2,3
3	1	COUT_1	'0'	4
4	1	SUM_1	'0'	2,3
5	1	COUT_2	'0'	4
6	1	SUM	'0'	
7	1	COUT	'0'	

Node 0 returns to line 13, reenters both the loop at line 13 and the loop at line 14. Behavioral Instance 0 is executed (an AND routine) with input signals 0 and 1.

After BI 0 is executed, a '0' at time 53 ns results for output signal 3, which connects to a BI on node 2. Per line 19, a message is sent to node 2, which posts the signal change to its Active Records. Likewise, a message is sent to node 1 for signal 4 changing to a '1' at 53 ns. After this,

node 0 continues through the loops and sends an "All Done" message to nodes 1 and 2, per line 49. This allows them to update their safe times for node 0 to MAX_TIME (2000 ns). The current status of each node is shown in Figures B.7 and B.8.

P0			P1			P2		
time	value	sigrec	time	value	sigrec	time	value	sigrec
			53	'1'	4	53	'0'	3

Figure B.7 Current Active Records.

P0	P1	P2
waiting[1] = T	waiting[0] = F	waiting[0] = F
waiting[2] = T	waiting[2] = T	waiting[1] = F
safe_time[1] = 2000	safe_time[0] = 2000	safe_time[0] = 2000
safe_time[2] = 2000	safe_time[2] = 2000	safe_time[1] = 6
sim_time = 2000	sim_time = 3	sim_time = 3

Figure B.8 Current Processor States.

Node 1's new Low_Time and Simulation_Time now become 53 ns. A null message is sent to Node 2 for 56 ns, as per line 35. This message allows node 2 to update its safe time for node 1 to 56 ns, and proceed. Since the signal on node 2 is still a '0', the Active Record is simply discarded per lines 38 and 39. Node 1's old value for signal 4 does not, however, equal the new value so the value is updated and the behavioral instances 2 and 3 (which reside on node 1) are added to the behavior list. Node 1's signal values are shown in Table B.6.

Node 1 reenters both loops at lines 13 and 14. Executing BI 2 results in a '0' for signal 5 at 56 ns. This information must be sent to node 2 since signal 5 is an input for BI

Table B.6 Node 1's Signal Records at Time 53 ns.

id	size	name	cur_val	conns
0	1	Y	'0'	0,1
1	1	X	'0'	0,1
2	1	CIN	'0'	2,3
3	1	COUT_1	'0'	4
4	1	SUM_1	'1'	2,3
5	1	COUT_2	'0'	4
6	1	SUM	'0'	
7	1	COUT	'0'	

number 4, which resides on node 2. A '1' results for signal 6 at 56 ns from executing BI 3. This signal has nothing in the "conns" field since it is a system output. Therefore, it is posted to node 1's own Active Records. The Active Records for each processor is shown in Figure B.9 and the simulation status is given in Figure B.10.

P0			P1			P2		
time	value	sigrec	time	value	sigrec	time	value	sigrec
			56	'1'	6	56	'0'	5

Figure B.9 Current Active Records.

P0	P1	P2
waiting[1] = T	waiting[0] = F	waiting[0] = F
waiting[2] = T	waiting[2] = T	waiting[1] = F
safe_time[1] = 2000	safe_time[0] = 2000	safe_time[0] = 2000
safe_time[2] = 2000	safe_time[2] = 2000	safe_time[1] = 53
sim_time = 2000	sim_time = 53	sim_time = 53

Figure B.10 Current Processor States.

Node 1 exits the loop at line 23, and updates its Low_Time and Simulation_Time to 56

ns. At line 35, it sends a null message for time 59 ns to node 2. That allows node 2 to update its safe time for node 1 at line 28. Node 2 then exits its loop at line 33, sees that the new value matches the old value for signal 5, and eventually goes back to waiting at line 26.

Node 1 enters the loop at line 36, sees that the old value does not match the new value and so updates its signal record. When a system output, such as "SUM" is updated, a message is printed for the user. Thus, the user would receive a message such as "SUM has changed from a '0' to a '1' at 56 ns." Node 1 now exits all of the loops and sends an "All Done" message to node 2, per line 49. Node 2 updates the safe time for node 1 to MAX_TIME, and since it has nothing left to do, exits the loops and finishes up as well. Now the system is quiesced and the simulation is over.

Appendix C. *Statistics*

This appendix contains the pertinent statistics for all of the test runs that were made during this research. The results and conclusions from these statistics are given in Chapters 5 and 6.

C.1 The Ripple-Carry Adder

Nodes	Mean	Std Dev	Min	Max	Loops	Vectors	Output
1	1695	37.864	1515	1747	0	32	No
1	1949	15.197	1908	2001	100	32	No
1	2954	23.084	2926	3026	500	32	No
1	4184	10.026	4151	4219	1000	32	No
1	6686	95.575	6629	7186	2000	32	No
1	14110	16.430	14076	14174	5000	32	No
1	26515	16.906	26476	26587	10000	32	No
1	4796	24.800	4762	4881	0	64	No
1	53201	15.033	53171	53253	10000	64	No
1	38959	5245.452	33096	55531	0	64	Yes
2	1076	14.019	1041	1093	0	32	No
2	1218	17.120	1185	1295	100	32	No
2	1731	34.941	1685	1874	500	32	No
2	2338	13.668	2309	2395	1000	32	No
2	3617	57.990	3551	3844	2000	32	No
2	7341	28.270	7303	7445	5000	32	No
2	13597	10.205	13565	13616	10000	32	No
2	2427	69.290	2394	2772	0	64	No
2	26713	10.932	26695	26754	10000	64	No
2	32670	2474.194	30258	38095	0	64	Yes
4	1182	13.594	1129	1211	0	32	No
4	1257	14.503	1222	1316	100	32	No
4	1530	56.459	1449	1728	500	32	No
4	1814	19.423	1751	1864	1000	32	No
4	2460	65.447	2412	2703	2000	32	No
4	4362	52.569	4299	4596	5000	32	No
4	7578	108.741	7493	8096	10000	32	No
4	2001	9.030	1982	2029	0	64	No
4	14262	13.865	14239	14316	10000	64	No
4	30797	583.105	30183	33138	0	64	Yes
8	1662	46.555	1520	1834	0	32	No
8	1887	266.764	1558	2712	100	32	No
8	1848	89.020	1680	2156	500	32	No
8	1980	15.967	1932	2040	1000	32	No
8	2409	183.300	2239	3019	2000	32	No
8	3322	144.604	3167	3930	5000	32	No

Nodes	Mean	Std Dev.	Min	Max	Loops	Vectors	Output
8	4851	14.304	4817	4871	10000	32	No
8	2200	7.214	2184	2214	0	64	No
8	8328	26.077	8248	8373	10000	64	No
8	32405	855.165	30413	33954	0	64	Yes

C.2 The Carry-Lookahead Adder

Nodes	Mean	Std Dev	Min	Max	Loops	Vectors	Output
1	10250	19.391	10220	10323	0	64	No
1	11180	18.045	11149	11260	100	64	No
1	14899	21.536	14871	14999	500	64	No
1	19551	16.753	19518	19623	1000	64	No
1	28847	19.017	28818	28914	2000	64	No
1	56747	15.501	56721	56809	5000	64	No
1	103245	35.570	103210	103427	10000	64	No
2	6725	14.292	6707	6772	0	64	No
2	7361	12.844	7337	7414	100	64	No
2	9871	11.633	9858	9927	500	64	No
2	13022	14.965	13004	13078	1000	64	No
2	19321	17.020	19302	19383	2000	64	No
2	38207	11.853	38190	38262	5000	64	No
2	69688	6.942	69672	69700	10000	64	No
4	3892	13.283	3864	3933	0	64	No
4	4298	27.163	4270	4295	100	64	No
4	5423	19.882	5396	5503	500	64	No
4	6927	13.976	6907	6974	1000	64	No
4	9935	12.985	9916	9969	2000	64	No
4	19004	13.583	18987	19065	5000	64	No
4	34148	24.107	34126	34263	10000	64	No
8	4338	53.422	4272	4528	0	64	No
8	4807	43.374	4721	4911	100	64	No
8	5133	26.722	5067	5183	500	64	No
8	5860	26.113	5776	5909	1000	64	No
8	7321	38.268	7224	7385	2000	64	No
8	12253	56.891	12153	12420	5000	64	No
8	20575	50.449	20426	20708	10000	64	No

C.3 The Carry-Save Adder

Nodes	Mean	Std Dev	Min	Max	Loops	Vectors	Output
1	4233	14.566	4208	4277	0	64	No
2	1932	68.958	1905	2288	0	64	No
4	1303	27.050	1286	1420	0	64	No
8	1426	17.145	1380	1442	0	64	No

Appendix D. *Configuration*

The purpose of this appendix is to show all of the files on the iPSC/2 hypercube that make up the parallel simulator and the three test cases. Currently, each test case uses its own directory. Therefore, some files may be redundant such as simutl.h. Some, however, are different even though the names are identical, such as queue8.dat. Therefore, the user must analyze the files, select the appropriate set of files, find the correct mapping for the simulation, sit back, and watch the simulation run!

D.1 The Files

File	Location	Purpose
cla.c	/usr2/eng/rcomeau/cla	Large C source file for VHDL behavior of carry-lookahead adder
csa.c	/usr2/eng/rcomeau/csa	Large C source file for VHDL behavior of carry-save adder
eba.c	/usr2/eng/rcomeau/eba	Large C source file for VHDL behavior of ripple-carry adder
host.c	/usr2/eng/rcomeau/cla	To load each node with node.c
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
node.c	/usr2/eng/rcomeau/cla	To run vhdl_main (in cla.c)
	/usr2/eng/rcomeau/csa	To run vhdl_main (in csa.c)
	/usr2/eng/rcomeau/eba	To run vhdl_main (in eba.c)
lp1.arcs	/usr2/eng/rcomeau/cla	Gives node dependencies for 1 node run
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
lp2.arcs	/usr2/eng/rcomeau/cla	Gives node dependencies for 2 node run
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
lp4.arcs	/usr2/eng/rcomeau/cla	Gives node dependencies for 4 node run
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
lp8.arcs	/usr2/eng/rcomeau/cla	Gives node dependencies for 8 node run
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"

File	Location	Purpose
queue1.dat	/usr2/eng/rcomeau/cla	Gives behavioral instance (gate) mapping for 1 node
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
queue2.dat	/usr2/eng/rcomeau/cla	Gives behavioral instance (gate) mapping for 2 nodes
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
queue4.dat	/usr2/eng/rcomeau/cla	Gives behavioral instance (gate) mapping for 4 nodes
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
queue8.dat	/usr2/eng/rcomeau/cla	Gives behavioral instance (gate) mapping for 8 nodes
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
makefile	/usr2/eng/rcomeau/cla	Makefile for carry-lookahead adder
	/usr2/eng/rcomeau/csa	Makefile for carry-save adder
	/usr2/eng/rcomeau/eba	Makefile for ripple-carry adder
pvsim.c	/usr2/eng/rcomeau/cla	Parallel VHDL SIMulator program. Switches for BUSY, OUTPUT, and MAPPING are here
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
runit1	/usr2/eng/rcomeau/cla	Gets cube of size 1 and runs host
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
runit2	/usr2/eng/rcomeau/cla	Gets cube of size 2 and runs host
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
runit4	/usr2/eng/rcomeau/cla	Gets cube of size 4 and runs host
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"
runit8	/usr2/eng/rcomeau/cla	Gets cube of size 8 and runs host
	/usr2/eng/rcomeau/csa	"
	/usr2/eng/rcomeau/eba	"

Appendix E. *Release*

This appendix contains a copy of an e-mail message from Doug Dunlop of Intermetrics Inc. whose help during this thesis is sincerely appreciated.

E.1 Authorization

```
From dunlop@inmet.camb.inmet.com Mon Nov  4 09:26:37 1991
To: rcomeau@galaxy
Subject: Re:  C source switch for VHDL

>   I am finalizing my thesis and was writing up how I parallelized VHDL.  Part
> of that process is capturing the C source code from the model generate stage
> using the "-debug=cknd" switch.  I need to know if that switch can be made
> available to NON-government agencies (i.e. is it proprietary?).  If you do
> not want that published, please let me know via e-mail or letter.  Thank you.

Including this information should be fine.  FYI the way of dumping the C
code has changed in recent versions so this information is not completely
current.  Good luck.

-- Doug
```

Bibliography

[Akl] Akl, Selim . *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, Inc., 1989.

[Bhas] Bhasker, J. "Algorithm for Microcode Compaction of VHDL Behavioral Descriptions," *MICRO: 19th Annual Microprogramming Workshop*: 54-58 (Dec 1987).

[Chan] Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM, 24*: 198-206 (April 1981).

[Coel] Coelho, David R. and Alec . Stanculescu. "A State of the Art VHDL Simulator," *1988 IEEE Spring COMPCON*: 320-323.

[Eick] Eickmeier, Daniel. *Software Requirements Validation of an SADT Specification Using Executable Simulation in VHDL*. MS thesis, A IT/ CS/EN /91D-06. School of Engineering, Air orce Institute of Technology, December 1991 (AD number not yet assigned).

[uji] ujimoto, Richard M. "Lookahead in Parallel Discrete Event Simulation," *Proceedings of the 1988 International Conference on Parallel Processing, 3*: 15-19 (1988).

[hos] hosh, Sumit and Meng-Lin Yu. "An Asynchronous Distributed Approach for the Simulation of Behavior-Level Models on Parallel Processors," *Proceedings of the 1988 International Conference on Parallel Processing, 1*: 74-77 (Aug 1988).

[las] lass, David N. "Compile-time Instruction Scheduling for Superscalar Processors," *Digest of Papers - Thirty-Fifth IEEE Computer Society International Conference COMPCON 89*: 630-633 (1990).

[uar] uarna, Vincent A., Jr. "A Technique for Analyzing Pointer and Structure References in Parallel Restructuring Compilers," *Proceedings of the 1988 International Conference on Parallel Processing, 3*: 212-220 (Aug 1988).

[Haye] Hayes, John P. *Computer Architecture and Organization*. Mc raw-Hill Book Company, 1988.

[VLRM] International Association of Electronics and Electrical Engineers, *VHDL Language Reference Manual*, 1987.

[Jeff] Jefferson, D. R. "Virtual Time," *ACM Transactions on Programming Languages and Systems, 7, 3*: 404-425 (1985).

[Koel] Koelbel, Charles and others. "Semi-automatic Process Partitioning for Parallel Computation," *International Journal of Parallel Programming, 16, 5*: 365-382 (1987).

[Lips] Lipsett, Roger and others. *VHDL: Hardware Description and Design*. oreword by Ronald Waxman. Norwell MA: Kluwar Academic Publishers, 1990.

[Midk] Midkiff, Samuel P. and David A. Padua. "Compiler Algorithms for Synchronization," *IEEE Transactions on Computers, C-36, 12*: 1485-1495 (Dec 1990).

[MILS] *Military Standard 454*, U.S. overnment Printing Office (1988).

[Nash] Nash, J. D. "Bibliography of Hardware Description Languages," *ACM SIGMA Newsletter, 14, 1*: 18-37 (eb 1984).

[Nico] Nicol, David M. "Analysis of Sychronization in Massively Parallel Discrete-Event Simulations," *SIGPLAN Notices, 25, 3*: 89-98 (Mar 1990).

[Poly] Polychronopoulos, Constantine D. "Compiler Optimizations for Enhancing Parallelism and Their Impact on Architecture Design," *IEEE Transactions on Computers, 37, 8*: 991-1003 (Aug 1988).

[Proi] Proicou, Michael Chris. *A Distributed Kernel for Simulation of the VHSIC Hardware Description Language*. MS thesis, A IT/ CS/EN /89D-14. School of Engineering, Air orce Institute of Technology, December 1989 (AD-A215419).

[Sart] Sartor, JoAnn M. *Optimal Iterative Task Scheduling for Parallel Simulations*. MS thesis, A IT/ CS/EN /91M-03. School of Engineering, Air orce Institute of Technology, March 1991 (AD-A238631).

[Shen] Shen, Zhiyu and others. "An Empirical Study of ortran Programs for Parallelizing Compilers," *IEEE Transactions on Parallel and Distributed Systems, 1, 3*: 356-364 (Jul 1990).

[Subr] Subramanian, K. and M. R. Zargham. "Distributed and Parallel Demand Driven Logic Simulation," *27th ACM/IEEE Design Automation Conference Proceedings*. 485-490. New York: Association for Computing Machinery, 1990.

[Tomb] Tombrello, Joseph and Ronald Turrentine. *C Language Based Fortran to Parallel Language Translator*. inal report, Contract No. DAS 60-89-C-0146, Sparta Inc., Huntsville, AL, ebruary 1989 (AD-B143 294L).

[Vyas] Vyas, M. C. and . N. Reddy. "A VHDL Based Design Environment for VLSI Circuits," *Proceedings of the 1989 IEEE SOUTHEASTCON '89, 2*: 401-405 (Apr 1989).

[Wait] Waite, Mitchell and Stephen Prata. *The Waite Group's New C Primer Plus*. Carmel, IN: Macmillan Computer Publishing, 1991.

[Waxm] Waxman, Ronald and Erich Marschner. "VHSIC Hardware Description Language (IEEE Standard 1076): Language eatures Revisited," *1988 IEEE Spring COMPCON*: 310-315 (1988).

[Wint] Winter, regory B. and Al Lowenstein. "Utilizing VHDL or Life Cycle Support of VHSIC-Class Systems," *IEEE 1986 AUTOTESTCON*: 145-151 (1986).

[Zima] Zima, Hans and Barbara Chapman. *Supercompilers for Parallel and Vector Compilers*. New York: ACM Press, 1990.